(12) **PATENT**
(19) **AUSTRALIAN PATENT OFFICE**

(11) Application No. **AU 200148024 B2**
(10) Patent No. **767293**

(54) Title
Image data acquisition optimisation

(51)[7] International Patent Classification(s)
G06T 011/00

(21) Application No: **200148024**

(22) Application Date: **2001.05.24**

(30) Priority Data

(31) Number	(32) Date	(33) Country
PQ7862	**2000.05.31**	**AU**
PQ7866	**2000.05.31**	**AU**
PQ7868	**2000.05.31**	**AU**

(43) Publication Date : **2001.12.06**
(43) Publication Journal Date : **2001.12.06**
(44) Accepted Journal Date : **2003.11.06**

(71) Applicant(s)
Canon Kabushiki Kaisha

(72) Inventor(s)
Ian Geoffrey Combes; Khanh Doan

(74) Agent/Attorney
SPRUSON and FERGUSON,GPO Box 3898,SYDNEY NSW 2001

(56) Related Art
AU 31348/01

S&F Ref: 545170

AUSTRALIA

PATENTS ACT 1990

COMPLETE SPECIFICATION

FOR A STANDARD PATENT

ORIGINAL

Name and Address of Applicant :	Canon Kabushiki Kaisha 30-2, Shimomaruko 3-chome, Ohta-ku Tokyo 146 Japan
Actual Inventor(s):	Alan Tonisson
Address for Service:	Spruson & Ferguson St Martins Tower,Level 35 31 Market Street Sydney NSW 2000
Invention Title:	Dynamically Pruned Compositing Expression Trees

ASSOCIATED PROVISIONAL APPLICATION DETAILS

[33] Country	[31] Applic. No(s)	[32] Application Date
AU	PQ6556	29 Mar 2000

The following statement is a full description of this invention, including the best method of performing it known to me/us:-

Abstract

DYNAMICALLY PRUNED COMPOSITING EXPRESSION TREES

A method and apparatus for creating an image is disclosed. The image is formed by rendering a plurality of graphical objects to be composited according to a hierarchical structure (500) representing a compositing expression for the image. The hierarchical structure (500) includes a plurality of nodes (e.g. 9) each representing a component of the image or an operation for combining sub-expressions of the compositing expression. The method comprises the steps of storing information for each node of the hierarchical structure (500), indicating at least an opacity of a sub-expression of the node. The stored information is dynamically updated for each node each time at least an opacity of a sub-expression of the node changes. At least one of the nodes is linked to form at least one circular list, depending on the stored information and the compositing expression is modified on the basis of the circular lists to form an optimised compositing expression for each of the sub-expressions. The image is composited using at least one of the optimised compositing expressions represented by the circular lists.

DYNAMICALLY PRUNED COMPOSITING EXPRESSION TREES

Technical Field of the Invention

The present invention relates generally to the creation of computer-generated images both in the form of still pictures and video imagery and, in particular, relates to an efficient process, apparatus, and system for creating an image made up by compositing multiple components. The invention also relates to a computer program product including a computer readable medium having recorded thereon a computer program for creating an image made up by compositing multiple components.

Background

Computer generated images are typically made up of many differing components or graphical objects which are composited together and rendered to create a final image. Each graphical object is generally represented as a fill type and a set of outlines. Graphical objects may be rendered by combining them onto a page using compositing operations. An "opacity channel" (also known as a "matte", an "alpha channel", or simply "opacity") is also commonly used as part of the representation of a graphical object. The opacity channel contains information regarding the transparent nature of each element. The opacity channel is typically stored alongside each instance of a colour, so that, for example, a pixel-based image with opacity stores an opacity value as part of the representation of each pixel. In this fashion each graphical object may be interpreted as occupying the entire image area, and the outlines of the graphical object can therefore define a region outside of which the object is fully transparent. This region is called the "active region" of the object. A graphical object without explicit opacity channel information is typically understood to be fully opaque within the outlines of the object, and assumed to be completely transparent outside those outlines.

One means for systematically representing an image in terms of its constituent elements and which facilitates in later rendering is an "expression tree". Expression trees typically comprise a plurality of nodes including leaf nodes, unary nodes and binary nodes. Nodes of higher degree, or of alternative definition may also be used. A leaf node, being the outer most node of an expression tree, has no descendent nodes and represents a primitive constituent of an image. Unary nodes represent an operation which modifies the pixel data coming out of the part of the tree below the unary operator. Unary nodes include such operations as colour conversions and convolutions (blurring etc). A binary node typically branches to left and right subtrees, wherein each subtree is itself an expression tree comprising at least one leaf node. Binary nodes represent an operation

which combines the pixel data of its two children to form a single result. For example, a binary node may be one of a number of standard compositing operators known in the art such as *over*, *in*, *out*, *atop* and alpha-*XOR*, examples of which and others are seen in Fig. 1. The compositing operations represented by the compositing operators are arithmetic or bitwise operations including opacity, which are performed on the component values of the colours of a graphical object .

Several of the above types of nodes may be combined to form the expression tree. An example of this is shown in Fig. 2. The result of the left-hand side of the expression tree illustrated in Fig. 2 may be interpreted as an image (Image 1) being colour (Conversion) clipped (In) to spline boundaries (Path). This construct is composited with a second image (Image 2).

Although the active region of a graphical object may of itself be of a certain size, it need not be entirely visible in a final image, or only a portion of the active region may have an effect on the final image. For example, assume an image of a certain size is to be displayed on a display. If the image is positioned so that only the top left corner of the image is displayed by the display device, the remainder of the image is not displayed. The final image as displayed on the display device thus comprises the visible portion of the image, and the invisible portion in such a case need not be rendered.

Another way in which only a portion of the active region of an object may have an effect is when the portion is obscured by another object. For example, a final image to be displayed (or rendered) may comprise one or more opaque graphical objects, some of which obscure other graphical objects. Hence, the obscured objects have no effect on the final image.

Two conventional rendering models include "frame buffer rendering" and "pixel sequential rendering". Conventional frame buffer rendering is based on the "painter's algorithm" which involves rendering multiple objects one at a time into a frame buffer. A compositing operation is carried out for every pixel within the boundaries of an object. A simple frame buffer approach is not sophisticated enough to render arbitrary expressions consisting of compositing operations involving transparency. Extra frame buffers may be required to render arbitrary expressions consisting of compositing operations involving transparency. The extra frame buffers are needed to store intermediate results.

Pixel sequential rendering as opposed to the "object sequential rendering" used in conventional frame buffer based systems involves performing all of the compositing operations required to produce a single pixel before moving on to the next pixel.

Interactive frame rates of the order greater than 15 frames per second can be achieved by such conventional methods, because the actual pixel operations are quite simple. Thus, conventional systems are fast enough to produce acceptable frame rates without requiring complex code. However, this is certainly not true in a compositing environment consisting of complex expression trees since the per-pixel cost of compositing is quite high. This is because typically an image is rendered in 24-bit colour in addition to an 8-bit alpha channel, thus giving 32 bits per pixel. Each compositing operator of an expression tree has to deal with each of the four channels.

Problems arise with the prior art methods in that they are highly inefficient since the per-pixel cost of evaluating the set of compositing operations required to compute each pixel is too high.

Disclosure of the Invention

It is an object of the present invention to substantially overcome, or at least ameliorate, one or more of the deficiencies of the above mentioned methods by the provision of a method of creating an image made up by compositing multiple components.

According to one aspect of the present invention there is provided a method of creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, comprising the steps of:

storing information for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

dynamically updating said stored information for each node each time at least an opacity of a sub-expression of said node changes;

linking at least one of said nodes to form at least one circular list, depending on said stored information;

modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each of said sub-expressions; and

compositing said image using at least one of said optimised compositing expressions represented by said circular lists.

According to another aspect of the present invention there is provided a method of creating an image, said image being formed by rendering at least a plurality of

graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein each of said objects has a boundary outline, said method comprising the steps of:

(a) storing information, for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

(b) linking at least one of said nodes to form at least one circular list, depending on said stored information;

(c) modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each said sub-expression;

(d) generating a pixel value for said image using at least one of said optimised compositing expressions represented by said circular lists;

(e) traversing said hierarchical structure and updating said information for each node, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of said sub-expressions changes; and

(f) repeating steps (b) to (e) at each object boundary outline.

According to still another aspect of the present invention there is provided a method of representing a compositing expression for an image using a hierarchical structure, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein nodes for a branch of said hierarchical structure are linked into a circular list, said circular list indicating an order in which said nodes are traversed to evaluate said compositing expression.

According to still another aspect of the present invention there is provided an apparatus for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, said apparatus comprising:

storage means for storing information for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

upate means for dynamically updating said stored information for each node each time at least an opacity of a sub-expression of said node changes;

linking means for linking at least one of said nodes to form at least one circular list, depending on said stored information;

modification means for modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each of said sub-expressions; and

compositing means for compositing said image using at least one of said optimised compositing expressions represented by said circular lists.

According to still another aspect of the present invention there is provided an apparatus for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein each of said objects has a boundary outline, said apparatus comprising:

storage means for storing information, for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

linking means for linking at least one of said nodes to form at least one circular list, depending on said stored information;

modification means for modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each said sub-expression;

pixel generation means for generating a pixel value for said image using at least one of said optimised compositing expressions represented by said circular lists; and

traverse means for traversing said hierarchical structure and updating said information for each node, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of said sub-expressions changes.

According to still another aspect of the present invention there is provided a computer readable medium, having a computer program recorded thereon, where the program is configured to make a computer execute a procedure for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, said program comprising:

code for storing information for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

code for dynamically updating said stored information for each node each time at least an opacity of a sub-expression of said node changes;

code for linking at least one of said nodes to form at least one circular list, depending on said stored information;

code for modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each of said sub-expressions; and

code for compositing said image using at least one of said optimised compositing expressions represented by said circular lists.

According to still another aspect of the present invention there is provided a computer readable medium, having a computer program recorded thereon, where the program is configured to make a computer execute a procedure for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein each of said objects has a boundary outline, said program comprising code for performing the following steps:

(a) storing information, for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

(b) linking at least one of said nodes to form at least one circular list, depending on said stored information;

(c) modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each said sub-expression;

(d) generating a pixel value for said image using at least one of said optimised compositing expressions represented by said circular lists;

(e) traversing said hierarchical structure and updating said information for each node, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of said sub-expressions changes; and

(f) repeating steps (b) to (e) at each object boundary outline.

According to still another aspect of the present invention there is provided a computer readable medium, having a computer program recorded thereon, where the program is configured to perform a procedure for representing a compositing expression for an image using a hierarchical structure, said hierarchical structure including a plurality

of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein nodes for a branch of said hierarchical structure are linked into a circular list, said circular list indicating an order in which said nodes are traversed to evaluate said compositing expression.

Brief Description of the Drawings

A number of preferred embodiments of the present invention will now be described with reference to the drawings, in which:

Fig. 1 shows the result of a variety of compositing operators useful with the present invention;

Fig. 2 is an example of a compositing tree;

Fig. 3 shows the region model used to define the Porter-Duff compositing operations;

Fig. 4 shows an expression tree with the instructions for evaluating the expression represented by the expression tree;

Fig. 5 shows an expression tree annotated with information about the opacity of each sub-expression in the expression tree, in accordance with a first embodiment of the present invention;

Fig. 6 shows the expression tree of Fig. 5 represented as an instruction table, in accordance with the first embodiment;

Fig. 7 is a flow chart showing how the opacity information for a tree is updated in accordance with the first embodiment;

Fig. 8 shows the expression tree of Fig. 5 after being updated in accordance with the first embodiment;

Fig. 9 shows an instruction table representing the expression tree of Fig. 8 in accordance with the first embodiment;

Fig. 10 shows a circuit for evaluating equations representing rules for evaluating Porter-Duff compositing expressions;

Fig. 11 shows a circuit that can be used to decide whether to prune the left or right sub-tree of a node, in accordance with the preferred embodiment;

Fig. 12 shows the expression tree of Fig. 5 modified in accordance with a second embodiment of the present invention;

Fig. 13 shows the expression tree of Fig. 12 modified in accordance with a second embodiment of the present invention;

Fig. 14 shows the expression tree of Fig. 13 modified in accordance with the second embodiment of the present invention;

Fig. 15 shows the expression tree of Fig. 14 modified in accordance with the second embodiment of the present invention;

Fig. 16 shows the expression tree of Fig. 15 modified in accordance with the second embodiment of the present invention;

Fig. 17 shows the expression tree of Fig. 16 modified in accordance with the second embodiment of the present invention;

Fig. 18 shows the expression tree of Fig. 17 modified in accordance with the second embodiment of the present invention;

Fig. 19 shows the expression tree of Fig. 18 modified in accordance with the second embodiment of the present invention;

Fig. 20 is a schematic block diagram of a general purpose computer upon which the preferred embodiment of the present invention can be practiced;

Fig. 21 is a flow chart showing the method of creating an image in accordance with the preferred embodiments of the present invention;

Fig. 22 is a flow chart showing the prune operation for disabling a parent-child relationship between two nodes of an expression tree, in accordance with the first embodiment;

Fig. 23 is a flow chart showing the graft operation for joining two branches of an expression tree, in accordance with the first embodiment;

Fig. 24 is a flow chart showing the skip operation for skipping redundant nodes, in accordance with the second embodiment;

Fig. 25 is a flow chart showing the unskip operation for re-inserting a node that has previously been skipped, in accordance with the second embodiment;

Fig. 26 is a flow chart showing a method for updating a pruned expression tree when a leaf node changes opacity, in accordance with the second embodiment;

Fig. 27 is a flow chart showing a method of accessing the parent node of a current node, in accordance with a further embodiment of the present invention;

Fig. 28 is a flow chart showing a process for updating the pruning of child nodes as a result of updating the opacity of the parent node, in accordance with the second embodiment; and

Fig. 29 is a flow chart showing a method of finding a handle for a given child node, in accordance with the second embodiment.

Detailed Description including Best Mode

Where reference is made in any one or more of the accompanying drawings to steps and/or features, which have the same reference numerals, those steps and/or features have for the purposes of this description the same functions or operations, unless the contrary intention appears.

Some portions of the detailed description which follows are explicitly or implicitly presented in terms of algorithms and symbolic representations of operations on data within a computer memory (ie. computer code). These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present description, discussions utilising terms such as "processing", "computing", "generating", "creating", "operating", "communicating", "rendering", "providing", and "linking" or the like, refer to the action and processes of a computer system, or similar electronic device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

In a typical image, most of the objects are either fully opaque or fully transparent at any given point on a page. Thus, there are generally very few objects contributing to any given pixel, typically only one or two, and rarely more than five. Therefore, in a complex image with a large number of objects, most of the compositing operations in an expression representing an image will be redundant.

The preferred embodiment is a method of creating an image. The preferred method is a more general and more powerful mechanism for reducing the number of pixel

compositing operations needed to calculate the colours of pixels and does not require extensive software analysis to generate input data. The image is formed by rendering graphical objects composited according to a hierarchical expression tree data-structure which represents a compositing expression for the image. The hierarchical tree data-structure includes nodes, each representing a component of the image or a Porter-Duff compositing operation for combining sub-expressions of the compositing expression. The nodes in the hierarchical tree preferably contain information about the opacity of sub-expressions including "pruning" information and "winding counts" for leaf nodes representing graphical objects. In accordance with the preferred embodiments, the opacity of sub-expressions of the expression is preferably dynamically updated and since expressions representing an image may change dynamically from transparent to indeterminate to fully opaque, a rule is provided to determine the opacity of sub-expressions.

Pruning information indicates relationships between sub-expressions. A node representing a sub-expression is considered to be "pruned" if the value at the node is not required to determine the value of the sub-expression represented by the parent node of the node, at the current pixel. Pruning is preferably represented, in accordance with the preferred embodiment, by linking nodes in circular lists. The preferred method generates instructions by traversing the tree and maintaining detailed information about which nodes to visit. The preferred method avoids the need to visit all of the nodes on each path from an active node to the root of the expression tree and eliminates the need for a stack for traversing the tree. The preferred method reduces the number of stack instructions required for generating expressions and makes the process of generating an instruction table simpler because there is no need for static opacity analysis by a compiler.

Fig. 21 is a flow chart showing the method of creating an image in accordance with the preferred embodiments. The process begins at step 2100, where the winding count, of each leaf node representing the image, is initialised and the opacity value of each node is set to transparent. Further, at step 2100, all connections between parent and child nodes are initialised to "pruned" which is represented by linking each node to itself (i.e. each node initially forms a circular list on its own). Also at step 2100, a scan conversion process of the image is commenced at the first pixel in the image area. As the scan proceeds from pixel to pixel, object boundaries of objects included in the image are checked, at step 2101, to determine if an object boundary has been crossed. If an object boundary has been crossed, the process continues at step 2103, where the opacity and

pruning information for the expression tree is updated, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of the sub-expressions changes. The process of updating the opacity and pruning information and the rules for determining whether the stored information is updated is described in more detail later in this document. If an object boundary was not crossed at step 2101, the tree does not need to be updated and a current list of rendering instructions is used to render the next pixel at step 2107. Each time the tree representing the image is updated, at step 2103, new rendering instructions are generated at step 2105 by traversing a circular list of links in the tree starting at the root node. Therefore, the compositing expressions represented by the tree and the circular lists are modified, based on the circular lists, to form an optimised compositing expression for each sub-expression. The process of traversing the circular list of links in the tree and generating new rendering instructions is described in more detail later in this document. Once the rendering instructions have been generated, the current pixel can be rendered at step 2107. After rendering each pixel of the image, the position of the scan is checked at step 2109 to determine if the end of the scan has been reached. If the end of the scan has been reached, the process concludes. Otherwise, the scan moves to the next pixel at step 2111 and then returns to step 2101.

The method of creating an image in accordance with the preferred embodiments will now be explained in more detail.

The compositing operators, discussed above, are defined in terms of a statistical model that assumes that each pixel has unit area, and is divided into four regions with areas determined by the opacities of the operands. If the left operand occupies a region L and the right operand occupies a region R, then the compositing operators may be defined in terms of the four combinations of L and R: $L \cap R$, $\bar{L} \cap R$ and $L \cap \bar{R}$ and $\bar{L} \cap \bar{R}$. These regions and their areas are shown in Fig. 3.

The compositing operators may also be defined in terms of four coverage flags: $C_{L\,R}$, $C_{R\,L}$, $C_{L\,\bar{R}}$ and $C_{R\,\bar{L}}$ whose meanings are defined below. If the opacities of the left and right operands are α and β, the areas of the regions L and R are assumed to be equal to α and β. If the colours of the left and right operands are A and B, respectively, then the colours of the regions are defined as follows. $C_{L\,R}$ indicates whether the left operand covers or obscures the right operand. If $C_{L\,R}$ is 1, then the colour of the region $L \cap R$ is A. $C_{L\,R}$ indicates whether the right operand covers or obscures the left operand. If $C_{R\,L}$ is 1, the colour of $L \cap R$ is assumed to be B. If $C_{L\,R}$ and $C_{R\,L}$ are both 0, then $L \cap R$ is

transparent. $C_{L\,R}$ and $C_{R\,L}$ are not both permitted to be 1. $C_{L\,\bar{R}}$ indicates whether the colour of the left operand is visible outside R. If $C_{L\,\bar{R}}$ is 1, then the colour of $L \cap \bar{R}$ is A, otherwise $L \cap \bar{R}$ is transparent. $C_{R\,\bar{L}}$ indicates whether the colour of the right operand is visible outside L. If $C_{R\,\bar{L}}$ is 1, the colour of $\bar{L} \cap R$ is B, otherwise $\bar{L} \cap R$ is transparent. Since $C_{L\,R}$ and $C_{R\,L}$ are not both permitted to be 1 at the same time, there are twelve possible compositing operators, of which nine are non-trivial.

Table 1, below, gives the definitions of all of the non-trivial Porter-Duff compositing operators in terms of the four coverage flags. All of the Porter-Duff compositing operators may be summed up in two equations. Equations (1) and (2) define the pre-multiplied colour C and the opacity O of the result of a compositing operation in terms of the four coverage flags.

$$C = C_{L\,R}\,\alpha\,\beta\,A + C_{L\,R}\,\alpha\,(1-\beta)\,A + C_{R\,L}\,\alpha\,\beta\,B + C_{R\,\bar{L}}\,(1-\alpha)\,\beta\,B \qquad (1)$$

$$O = C_{L\,R}\,\alpha\,\beta + C_{L\,\bar{R}}\,\alpha\,(1-\beta) + C_{R\,L}\,\alpha\,\beta + C_{R\,\bar{L}}\,(1-\alpha)\,\beta \qquad (2)$$

Operator	$C_{L\,R}$	$C_{L\,\bar{R}}$	$C_{R\,L}$	$C_{R\,\bar{L}}$
Over	1	1	0	1
Rover	0	1	1	1
In	1	0	0	0
Rin	0	0	1	0
Out	0	1	0	0
Rout	0	0	0	1
Atop	1	0	0	1
Ratop	0	1	1	0
Xor	0	1	0	1

Table 1

Compositing expressions can be evaluated using a simple stack based processor with 10 instructions. Table 2, below, gives a minimal set of stack instructions for evaluating Porter-Duff compositing operations. Each instruction is described in terms of its effect on the stack. Preferably, a pixel generator is provided, in accordance with the preferred embodiment, to supply the current pixel on demand for any graphical object in

the expression tree. The PUSH instruction fetches the current pixel for a given graphical object from the pixel generator. The preferred rendering model also includes other bitwise operations to support GDI rendering and instructions that are equivalent in action to a PUSH followed by one of the compositing instructions as well as other stack operations. The instruction set presented in this document is simplified to illustrate the method of creating an image without making the description unnecessarily complicated.

Instruction	Description
OVER	Pop A, pop B, push (A over B)
ROVER	Pop A, pop B, push (A rover B)
IN	Pop A, pop B, push (A in B)
RIN	Pop A, pop B, push (A rin B)
OUT	Pop A, pop B, push (A out B)
ROUT	Pop A, pop B, push (A rout B)
ATOP	Pop A, pop B, push (A atop B)
RATOP	Pop A, pop B, push (A ratop B)
XOR	Pop A, pop B, push (A xor B)
PUSH G	Push current pixel from gob G

Table 2

Fig. 4 shows an expression tree 400 with the instructions for evaluating the expression. The instructions for a given expression can be generated by performing a right to left post-order traversal of the expression tree. In Fig. 4, the nodes are numbered in the order that they are visited in the traversal. For each graphic object encountered, a push instruction is generated. For each compositing operator encountered, the corresponding compositing instruction is generated.

In an expression composed of compositing operators, it is possible to determine at each pixel position on the page, for each sub-expression, whether or not the sub-expression represents an opaque colour, a transparent colour or a colour that may have some other opacity. Given that it is known whether the operands are opaque, transparent or of indeterminate opacity, in most cases, it is possible to determine the result of an operation without needing to perform any colour calculations.

Tables 3 to 11, below, give rules for evaluating compositing operations given knowledge about the opacity of the operands. In each table, the first two columns give all possible combinations of opacity information for the operands of a compositing operation. A 'T' indicates that the operand is known to be transparent, an 'O' indicates that the operand is known to be opaque, and a '?' indicates that the operand has indeterminate opacity. The third column gives an expression equivalent to the value of the compositing expression and the fourth column indicates the opacity of the result of the compositing operation. A dash in the third column indicates that the result is fully transparent, so neither of the operand colours contribute to the result. In most cases, the

resulting colour is either transparent or equal to the colour of one of the operands; in these cases, no stack operation needs to be generated. The fifth column indicates which sub-trees should be pruned. The sixth column indicates the stack operation that needs to be generated to evaluate the operation. A dash indicates that no instruction needs to be generated.

By applying the rules in the tables 3 to 15 to maintain information about the opacity of each sub-expression in a compositing expression, the embodiments of the present invention reduce the number of pixel compositing operations that need to be performed to evaluate the expression. For example, given an expression of the form A rin B, where the sub-expression A is known to be fully opaque, rows 4, 5 and 6 of Table 6 indicate that the sub-expression A does not need to be evaluated at all.

A	B	A over B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	B	O	Left	-
T	?	B	?	Left	-
O	T	A	O	Right	-
O	O	A	O	Right	-
O	?	A	O	Right	-
?	T	A	?	Right	-
?	O	A over B	O	None	OVER
?	?	A over B	?	None	OVER

Table 3

A	B	A rover B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	B	O	Left	-
T	?	B	?	Left	-
O	T	A	O	Right	-
O	O	B	O	Left	-
O	?	A rover B	O	None	ROVER
?	T	A	?	Right	-
?	O	B	O	Left	-
?	?	A rover B	?	None	ROVER

Table 4

A	B	A in B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	-	T	Both	-
T	?	-	T	Both	-
O	T	-	T	Both	-
O	O	A	O	Right	-
O	?	A in B	?	None	IN
?	T	-	T	Both	-
?	O	A	?	Right	-
?	?	A in B	?	None	IN

Table 5

A	B	A rin B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	-	T	Both	-
T	?	-	T	Both	-
O	T	-	T	Both	-
O	O	B	O	Left	-
O	?	B	?	Left	-
?	T	-	T	Both	-
?	O	A rin B	?	None	RIN
?	?	A rin B	?	None	RIN

Table 6

A	B	A out B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	-	T	Both	-
T	?	-	T	Both	-
O	T	A	O	Right	-
O	O	-	T	Both	-
O	?	A out B	?	None	OUT
?	T	A	?	Right	-
?	O	-	T	Both	-
?	?	A out B	?	None	OUT

Table 7

A	B	A rout B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	B	O	Left	-
T	?	B	?	Left	-
O	T	-	T	Both	-
O	O	-	T	Both	-
O	?	-	T	Both	-
?	T	-	T	Both	-
?	O	A rout B	?	None	ROUT
?	?	A rout B	?	None	ROUT

Table 8

5

A	B	A atop B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	B	O	Left	-
T	?	B	?	Left	-
O	T	-	T	Both	-
O	O	A	O	Right	-
O	?	A atop B	?	None	ATOP
?	T	-	T	Both	-
?	O	A atop B	O	None	ATOP
?	?	A atop B	?	None	ATOP

Table 9

A	B	A ratop B	Opacity	Instruction
T	T	-	T	-
T	O	-	T	-
T	?	-	T	-
O	T	A	O	-
O	O	B	O	-
O	?	A ratop B	O	RATOP
?	T	A	?	-
?	O	A ratop B	?	RATOP
?	?	A ratop B	?	RATOP

Table 10

A	B	A xor B	Opacity	Prune	Instruction
T	T	-	T	Both	-
T	O	B	O	Left	-
T	?	B	?	Left	-
O	T	A	O	Right	-
O	O	-	T	Both	-
O	?	A xor B	?	None	XOR
?	T	A	?	Right	-
?	O	A xor B	?	None	XOR
?	?	A xor B	?	None	XOR

Table 11

Fig. 5 shows an example of a compositing expression tree 500 annotated with information about the opacity of each sub-expression in the expression tree 500, in accordance with a first embodiment of the present invention. Each node in the tree 500 is annotated with a code indicating the opacity of the sub-expression that the node is the root of. Each node in the tree is also numbered 1 through 8 for illustrative purposes. Straight lines (e.g. 501) connect each node (e.g. 505) to its parent node (e.g. 507). Dashed lines (e.g. 503) indicate that a sub-expression's colour and opacity is not required to calculate the colour and opacity of its parent node. The dashed lines indicate where a *connection* in the tree has been "pruned". A pruned *connection* indicates that a sub-expression is temporarily not required for generating instructions for the parent sub-expression. A pruned link indicates that a sub-expression is temporarily not required for generating

instructions for the sub-expression in which its parent is contained. A node is said to be pruned if the link between it and its parent is pruned.

Deleting the pruned connections partitions the expression tree 500 into disjoint sub-trees. These sub-trees are called "branches". Sub-trees of branches are called "sub-branches". The tree 500 shown in Fig. 5 is split into four branches. Nodes 6 and 7 together form a branch, nodes 1, 3, 4 and 9 form a branch, and nodes 5 and 8 each form a branch on their own.

In accordance with the first embodiment, the information about pruning is maintained by linking the nodes in each branch into a circular list. The links in the circular lists are indicated by arrows (e.g. 515). Each circular list is preferably linked in right to left post-order traversal order with the root node linked to the first node in the traversal. The traversal order is the order in which the nodes need to be visited to generate stack instructions to evaluate the sub-expression represented by a branch containing the nodes.

Fig. 6 shows the expression tree 500 shown represented as an instruction table, in accordance with the first embodiment. The preferred instruction table data-structure consists of records such that there is preferably one record per node in an expression tree represented by the table. Each row in Fig. 6 represents one record containing data pertaining to the node and pointers to other related nodes. A pointer consists of the index of a record.

The records in the preferred instruction table are stored in such an order that each sub-expression is stored in a contiguous block of memory. The root node is stored first, followed by the left sub-tree if there is one, followed by the right sub-tree if there is one; each sub-expression being represented recursively in a similar manner.

Each record in the preferred instruction table contains twelve fields. The OP field represents the operation or graphical object that the node represents. The FILL RULE and WINDING COUNT fields together determine whether a leaf node representing a graphical object is active or not. O_L, A_L, O_R, A_R, O_{OP} and A_{OP}, represent the opacities of the node and its children. The PARENT and END fields are used to encode the structure of the unpruned expression tree. The NEXT field is used to represent the pruning of the tree and to generate instructions. The meanings of each of the fields will be explained in more detail below.

The OP field represents the operation to perform and is used to determine which stack instruction to generate for each node. For leaf nodes, the operation will be a push of

the current pixel of a graphical object. The OP field of a leaf node preferably includes the index of a "fill" that is used to get or generate the pixel. As seen in Fig. 6, the push operations are denoted by a letter indicating a fill to use to generate an associated pixel.

The O_{OP} and A_{OP} fields indicate the opacity of the result of the operation at a current pixel position. A_{OP} is 1 when the node is active and 0 if not (i.e. when the current pixel is not known to be transparent). O_{OP} is 1 if a node is known to be opaque when it is active and 0 otherwise. Table 12, below, shows how the fields encode the opacity of an associated node.

A_{OP}	O_{OP}	Opacity
0	0	T
0	1	T
1	0	?
1	1	O

Table 12

For leaf nodes, O_{OP} can be determined statically by a compiler or it can be generated dynamically depending on how the pixels are generated for each object. Preferably, the values of the O_{OP} flag for leaf nodes are supplied by a compiler producing an associated image. The O_{OP} flag provides a mechanism for obscurance optimisation. For each non-leaf node, the O_{OP} flag is calculated dynamically from the opacities of an associated node's children. O_{OP} is ignored if the node is not active (i.e. when A_{OP} is 0), so there are two ways provided to encode a transparent pixel.

For non-leaf nodes, O_L and A_L are copies of the O_{OP} and A_{OP} flags, respectively, of the root of the node's left sub-tree, and O_R and A_R are copies of the O_{OP} and A_{OP} flags, respectively, of the root of the node's right sub-tree. The O_L, A_L, O_R and A_R flags are used only to avoid the need to load the root records of both of the sub-trees when the opacity needs to be updated. For leaf nodes, O_L, A_L, O_R and A_R are unused. Note that for non-leaf nodes O_{OP} and A_{OP} can be calculated from the other flags, so there is no need to store them. Therefore, the record sizes can preferably be reduced by two bits with no loss of information, but this optimisation will be ignored in the following description to keep the description simple.

The preferred data-structure allows the tree to be efficiently traversed up and down. The PARENT pointers are preferably used to traverse up an associated tree from the leaves towards the root. Given a pointer to an operation node, incrementing the pointer by one gives a pointer to the root of a left sub-tree of the node. The END field is a pointer indicating the end of the sub-expression that the node is the root of. The END

field points to the next record location after the last record in the sub-expression. The root of right sub-tree can be found by first obtaining a pointer to the root of the left sub-tree and looking up the END field in that record.

The preferred data-structure can be traversed efficiently in the order required to generate stack instructions for the expression. The NEXT field is a list pointer in the circular list linking the nodes in a branch to which the node belongs. The NEXT field is used to encode the pruning of an expression tree and to generate instructions. Instructions can be generated by traversing the NEXT fields starting from the root of the expression tree.

The FILL RULE field preferably stores the rule used to determine the "active region" of the object from its boundaries, in accordance with the first embodiment of the present invention. The fill rule can be non-zero winding, odd/even or winding counting. The WINDING COUNT field is preferably initialised to 0 at the start of a render, and is updated each time the boundary of an object is crossed. Object boundaries are represented by edges that have an orientation that determines whether to increment or decrement the winding count when an edge is crossed. The fill rule determines how to interpret the winding count. An object's "active region" is determined by the edges defining its boundary and by its fill rule.

In accordance with the first embodiment, the PARENT and END fields plus the ordering of records are preferably used to represent the structure of an associated expression tree. Each successive argument of an operator can be found by de-referencing one END pointer.

The maintenance of the opacity information and the maintenance of the pruning information, in accordance with the first embodiment, are independent except that the decision to prune or restore the connections between any node and it's child nodes is determined by changes to the opacity of the child nodes. The following description details how the opacity information is maintained while ignoring the pruning information. In accordance with the first embodiment, the pruning operations are combined with the update of the opacity information to reduce the number of times that records need to be accessed.

Fig. 7 is a flow chart showing how the opacity and pruning information for an expression tree is updated, as at step 2013, in accordance with the first embodiment. The process begins at step 701, where a node representing an object whose boundary has been crossed by the scan is selected as the current node. At the next step 703, the winding

count for the node is updated in accordance with the orientation of the crossed boundary. The process continues at step 705, where the opacity value stored at the node is updated. The updated opacity is determined by the node's fill rule in conjunction with the updated value of the winding count. The process of determining the updated opacity is described in more detail later in this document. At the next step 707, the previous value of the opacity is compared with the new value. If the new opacity value is the same as the old opacity value, the process concludes. Otherwise, if the new opacity value is different from the old opacity value, the process continues to step 709, where the ancestors of the leaf nodes need to be updated. At the next step 709, a test is performed to determine if the current node is the root of the expression tree. If the current node is the root of the expression tree, all ancestors of the leaf node have been updated. Otherwise, if the current node is not the root node, the parent of the current node is made the new current node at step 711. The process then continues at step 713 where the pruning information about the connections between the current node and the current node's children is updated. The process of updating the pruning information is described in more detail later in this document. The process then returns to step 705, where the opacity information is updated for the current node.

Preferably, edge crossing information is automatically generated in accordance with the embodiments of the invention. Pixels are preferably generated in scan line order, and each time an object boundary is encountered by a scan line, an opacity update command will be generated that indicates the change in opacity of the given object. Such a command preferably contains the index of the node in the expression table, and the new opacity of the node.

Each time the edge of an object is encountered, the WINDING COUNT field of the instruction table associated with the object is preferably updated to reflect the change and an opacity update command is generated to update the instruction table. For example, given the instruction table shown in Fig. 6, a downward edge crossing of object 3, which is associated with node 3 of the expression tree 500, will reduce the winding count in row 8 of the instruction table from 2 to 1. Since the fill rule in row 8 is odd/even, the object associated with node 3 becomes active and hence opaque, so an opacity update will preferably be generated such that instruction 8 has an opacity value of O. Instruction 8 is the right argument of the out operator, representing fill D. To update the entire tree, the parent of node 8 (i.e. node 6) representing an out operation, also needs to be updated. Row 8 of Table 7 indicates that the new opacity of node 6 is T, and hence the left operand

is no longer needed and can be pruned. Since the opacity of node 6 has changed, it's parent (i.e. node 2) also needs to be updated. Row 4 of Table 3 indicates that the opacity of node 2 should be O. Since this opacity is the same as before, there is no need to update the parent of node 2. The updated expression tree 800 resulting from the update command is shown in Fig. 8 and the updated instruction table representing the tree 800 is shown in Fig. 9.

Preferably, a leaf node of an expression tree is updated whenever it's activity state changes, and whenever a node changes opacity, it's parent node's opacity is preferably updated. The process stops when a node does not change opacity. Each time a node is updated, the links representing the pruning of the associated tree need to be updated as well. The preferred method of updating these links will be described in more detail below.

To avoid the need to store tables 3 to 11, the rules in these tables can be expressed in terms of the four coverage flags: C_{LR}, C_{RL}, $C_{L\bar{R}}$ and $C_{R\bar{L}}$ that define the Porter-Duff compositing operators. A_{OP} and O_{OP} can be calculated as follows:

$$A_{OP} = (C_{LR} \vee C_{RL})A_L A_R \vee C_{L\bar{R}} A_L \overline{(A_R O_R)} \vee C_{R\bar{L}} A_R \overline{(A_L O_L)} \quad \ldots\ldots (3)$$

$$O_{OP} = (C_{LR} \vee C_{RL})A_L O_L A_R O_R \vee A_L O_L C_{L\bar{R}} (C_{LR} \vee \overline{A_R}) \vee A_R O_R C_{R\bar{L}} (C_{RL} \vee \overline{A_L}) \quad \ldots\ldots (4)$$

There are two decisions that need to be determined from tables 3 to 11. Firstly, whether or not the left and right operands need to be pruned and secondly, whether or not a stack operation needs to be generated for the node. Since a stack operation only needs to be performed when neither node is pruned, the determination of whether to generate a stack operation can be made from the pruning information.

The left sub-tree of a Porter-Duff compositing expression tree can be pruned precisely when the expression evaluates either to transparent or to the value of the right sub-expression. Any other value requires knowledge of either the colour or alpha value of the left sub-tree. Similarly, the right sub-tree can be pruned when the value of the expression is either transparent or equal to the value of the left sub-expression.

Letting P_L be a boolean decision variable which is true if and only if the left sub-tree of an associated expression tree should be pruned and letting P_R be a decision variable which is true if and only if the right sub-tree should be pruned. The values of these decision variables can be expressed in terms of the instruction table flags as follows:

$$P_L = \overline{A_L} \vee \overline{C_{LR} A_R} (\overline{C_{L\bar{R}}} \vee A_R O_R)(\overline{C_{RL} \oplus C_{R\bar{L}}} \vee O_L \vee \overline{A_R}) \quad \ldots\ldots (5)$$

$$P_R = \overline{A_R} \vee \overline{C_{RL} A_L} (\overline{C_{R\bar{L}}} \vee A_L O_L)(\overline{C_{LR} \oplus C_{L\bar{R}}} \vee O_R \vee \overline{A_L}) \quad \ldots\ldots (6)$$

Equation 5 is derived from equation 1. To prune the left sub-tree, there must be no contribution from the colour of the left sub-expression (i.e. $C_{L\,R}\,\alpha\,\beta + C_{L\,R}\,\alpha\,(1-\beta) = 0$), and the premultiplied colour of the compositing operation C_{OP} must evaluate either to 0 or to the premultiplied colour of the right sub-expression (i.e. $C_{L\,R}\,\alpha\,\beta + C_{L\,R}\,\alpha\,(1-\beta) = 0$ and either $\beta = 0$ or $C_{R\,L}\,\alpha + C_{R\,L}\,(1-\alpha) = 0, 1$).

An instruction only needs to be generated for the node when P_L and P_R are both false.

Two operations are used to update the branch lists, in accordance with the first embodiment. The operations are called "`prune`" and "`graft`". The `prune` operation disables a parent-child relationship between two nodes in a tree and the `graft` operation restores a parent-child relationship between two nodes. `prune` splits a branch into two branches and `graft` joins two branches together. These operations each take one parameter, that is a pointer to the child node of the link to be disabled or restored.

`prune` and `graft` are inverses of each other. If a node `n` is not pruned, then `prune n` followed by `graft n`, will result in no change. Similarly, if a node `n` is pruned, then graft n followed by prune n will result in no change.

The following pseudo C code shows the preferred algorithm for the prune and graft operations in accordance with the first embodiment. The code uses C pointer notation for record indices/addresses and uses pointer de-referencing to access fields in each record.

```
void prune(Node * child)
{
    // Find start of branch's list.
    Node * n = child->PARENT;
    while (n->NEXT < n)
        n = n->NEXT;
    // Find the entry point into the child subtree.
    while (n->NEXT ≥ child->END)
        n = n->NEXT;
    // swap n->NEXT and child->NEXT
    Node * t = n->NEXT;
    n->NEXT = child->NEXT;
    child->NEXT = t;
}

void graft(Node * child)
{
    // Find start of parent's branch's lists.
    Node * n = child->PARENT;
    while (n->NEXT < n)
        n = n->NEXT;
    // Find where to insert the child subtree.
    while (n->NEXT ≥ child->END)
        n = n->NEXT;
    // swap n->NEXT and child->NEXT
    Node * t = n->NEXT;
    n->NEXT = child->NEXT;
    child->NEXT = t;
}
```

The prune and graft operations consist of two parts, finding either the insertion point or the entry point for a sub-branch followed by a pointer swap. The entry point of a sub-branch is the node that precedes it in a list linking the nodes in the branch of the sub-branch. The insertion point of a branch is the node that precedes the sub-tree in the list of

nodes linking the branch containing the parent of a pruned sub-tree. Note that the prune algorithm is identical to the graft algorithm.

Since entry and insertion points have so much in common, it is useful to blur the distinction between them by defining a term to refer to both types of nodes. Entry points and insertion points are both referred to as "linkage points". That is, if a sub-tree is pruned, its linkage point is its insertion point otherwise its linkage point is its entry point. The linkage point of a node is defined to be the linkage point of the pruned sub-tree that the node is the root of. Thus, every node in an expression tree except the root has a linkage point. For example, in Fig. 8, node 9 is the linkage point for nodes 2, 3, 4, 5 and 6, and node 6 is the linkage point for nodes 7 and 8.

Fig. 22 is a flow chart showing the `prune` operation for disabling a parent-child relationship between two nodes of an expression tree, in accordance with the first embodiment. Fig. 22 corresponds to the above pseudo C code showing the preferred algorithm for the `prune` operation. The process of Fig. 22 begins at step 2201, where the root of the node to be pruned is determined. At the next step 2203, the entry point into the child sub-tree for the root of the node is determined. The process of Fig. 22 concludes at the next step 2205, where the child sub-tree is unlinked from the node in accordance with the first embodiment.

Fig. 23 is a flow chart showing the `graft` operation for joining two branches of an expression tree, in accordance with the first embodiment. Fig. 23 corresponds to the above pseudo C code showing the preferred algorithm for the `graft` operation. The process of Fig. 23 begins at step 2301, by determining the root of the parent node to which the child sub-tree is to be joined. At the next step 2303, the point at which to insert the child sub-tree is determined. The process of Fig. 23 concludes at the next step 2305, where the child sub-tree is linked to the parent node in accordance with the first embodiment.

The algorithms for `prune` and `graft` shown in the above pseudo C code and in Figs. 22 and 23, both involve a sequential search of a branch for each operation. There are two ways to improve the algorithm. Firstly, avoid searching wherever possible, and secondly, reduce the size of the searches that must be done. In accordance with another embodiment of the present invention, searching is avoided by caching results of previous searches. There are various ways of reducing the size of the searches, but these result in more complicated algorithms for `prune` and `graft`. It is better to modify the data structure to skip redundant nodes and keep the algorithms as simple as possible. An

improvement to the data-structure of the first embodiment that reduces the size of the searches by skipping redundant nodes is described below.

The cost of pruning and grafting operations can be reduced by being aware of the sequence in which these operations are generated as a result of the opacity update operations. Opacity update operations preferably start at a leaf node and proceed up the associated tree towards the root. Opacity update operations only affect nodes that are on a path from an updated leaf node towards the root of the tree, or nodes that are immediately adjacent to such a path. In accordance with another embodiment, linkage points can be re-used as the update propagates up the tree.

Table 13, below, shows the linkage points of the child nodes of a non-leaf node denoted by "parent". The non-leaf node's two child nodes are denoted by "left" and "right". The linkage points are determined by whether or not the parent node is pruned and whether or not the right sub-tree is pruned. If the parent is pruned, then the linkage points of the child nodes can be determined without searching. If the parent is pruned, the linkage point of the parent can be the same as the linkage point of one or both of the parent nodes children nodes, so linkage can sometimes be determined without searching. Preferably searching is avoided by calculating a parent's linkage point from its child each time an associated update progresses up the tree. The linkage points of the child nodes can be determined from the available information when possible.

The following pseudo C code shows the preferred method for caching linkage points. The code uses C pointer notation for record indices/addresses and uses pointer de-referencing to link fields in each record. The function `getParentNode` shows how to update the cached linkage point. The `getParentNode` function gets the parent node of the current node and either keeps the current linkage point if it is the same as the child's linkage point or sets the current linkage point to `NULL` if not. The code for `getParentNode` assumes that the parent nodes opacity and pruning flags have been updated already. The functions `getLeftLinkagePoint` and `getRightLinkagePoint` return the addresses of the linkage points of the left and right child nodes respectively of the current node.

The following pseudo C code also shows the preferred algorithm for caching addresses in accordance with a further embodiment. Alternatively, the records themselves can be cached instead of or in addition to their addresses. The code relies on several helper functions `isPruned`, `getLeft`, `getRight` and `findLinkagePoint`. `isPruned`, returns true if the given node is pruned.

`findLinkagePoint` searches for the current node's linkage point and updates the variable `linkagePoint`. `getLeft` and `getRight` return the addresses of the roots of the left and right sub-trees of a node, respectively.

```
Node * current;       // currrent node
Node * linkagePoint; // linkage point for current node

void getParentNode()
{
    // Remember the child node.
    Node * child = parent;
    // Get new current node.
    current = child->PARENT;     // get current node
    // If current node's linkage point is not the same
    // as the child's, then set linkagePoint to NULL.
    if ( isPruned(current) ||
            (!(current->PR)       &&       (child       ==
getLeft(current))) )
        linkagePoint = NULL;
}

Node * getLeftLinkagePoint()
{
    if (isPruned(current))
    {
        if (current->PR)
            return current;
        else
            return getRight(current);
    }
    else
    {
        if (current->PR)
        {
            if (linkagePoint == NULL)
                findLinkagePoint();
            return linkagePoint;
        }
```

```
            else
                return getRight(current);
        }
    }

    Node * rightLinkagePoint()
    {
        if (isPruned(current))
            return current;
        else
        {
            if (linkagePoint == NULL)
                findLinkagePoint();
            return linkagePoint;
        }
    }
```

Fig. 27 is a flow chart showing a method of accessing the parent node of the current node, in accordance with a further embodiment of the present invention. The method of Fig. 27 updates all cached node addresses when moving from a node to the parent of the node after the node has been updated. Fig. 27 corresponds to the above pseudo C code showing the preferred `getParentNode` algorithm for updating a cached linkage point. The process of Fig. 27 begins at step 2701, where a current node flag (i.e. *current*) representing the current node being processed, is set to the address of the parent node of the current node. At the next step 2703, the cached linkage point associated with the parent node is updated such that if the linkage point of the parent is not the same as the linkage point of the child then the current linkage point is set to NULL. Alternatively, the parent node keeps the same linkage point if the linkage point of the parent is the same as the linkage point of the child. The process of Fig. 27 concludes once the cached linkage point associated with the parent node of the current node has been updated.

To access records efficiently, four nodes are preferably cached: the current node, its left and right child nodes and the linkage point for the current node. These nodes should be loaded when needed and cached for subsequent operations.

A data structure in accordance with a second embodiment of the present invention will now be described. The data structure of the second embodiment further reduces the amount of work required to search for linkage points. At the cost of a small amount of maintenance overhead, the second embodiment significantly reduces the amount of work required to generate instructions to composite pixels for typical images.

The data-structure of the second embodiment works on the principle that if a node has one unpruned sub-tree, the node does not contribute an operation to the list of stack instructions. In accordance with the second embodiment nodes with one unpruned sub-tree can be ignored by unlinking them from the list of nodes representing a branch. It is important to be able to find the root node of each branch, so branch roots must remain linked. Therefore, all of the nodes in each branch are linked into a circular list except non-root nodes with exactly one unpruned sub-tree. That is, a node is linked into the circular list if and only if either the node is the root of a branch or the operation that the node represents is required to evaluate the expression represented by the branch. The data structure of the second embodiment results in reduced update times as well as reducing the time to generate lists of stack instructions. Non-root nodes with only one sub-tree have null NEXT pointers.

For typical images, the method of the second embodiment will result in very short branch lists since long lists will only appear in an image with many translucent objects that are all simultaneously active. Such images are not very common. In an image containing only objects that are known not to be translucent anywhere, these lists will never contain more than two nodes.

Fig. 12 shows an expression tree 1200 which is the same expression tree as shown in Fig. 5, with the NEXT pointers modified in accordance with the second embodiment. Note that nodes 3, and 2 are no-longer linked into the circular list representing the branch rooted at node 1 since they do not contribute to the list of stack instructions needed to evaluate the branch.

In the expression tree 1200 shown in Fig. 12, if node 4 becomes inactive (i.e. transparent), the sequence of pruning update operations will be: `prune 4`, `prune 3`, `graft 6`. Unfortunately, since the data structure of the second embodiment is different, the above described methods of `prune` and `graft` will not work. In accordance with the second embodiment, two new helper operations are defined: `unskip` and `skip`. The two new operations are used to add and remove redundant nodes from the list of nodes belonging to a branch, redundant nodes being nodes with exactly one sub-tree. The

`unskip` and `skip` operations convert parts of the tree between the unoptimised and optimised versions of the data-structure. The `unskip` and `skip` operations each take one argument representing a node to be removed or inserted into the list of nodes in a branch. The `skip` and `unskip` operations are introduced in order to assist in describing how the data structure is maintained, in accordance with the second embodiment. In accordance with a further embodiment, the `skip` and `unskip` operations can be combined with the pruning and grafting operations.

Figs. 13 to 15 each show the result of a step in updating the expression tree 1200 of Fig. 12 resulting from a change of object 4 from opaque to transparent, in accordance with the second embodiment. In Figs. 13 to 15, the nodes affected in a particular step are highlighted with shading. Before node 4 can be pruned, its parent (i.e. node 3), must be unskipped and the result of this operation is shown in Fig. 13. Fig. 14 shows the result of pruning node 4. Note that pruning node 4 leaves the tree in a state that is not possible for a fully updated tree since in a fully updated tree, every leaf of a branch is a leaf of the original expression tree. In the pruned expression tree shown in Fig. 14, node 3 is a leaf of the branch with root 1, but it is not a leaf of the unpruned expression tree 1200. Before node 3 can be pruned, node 2 must be unskipped. The result of the unskipping of node 3 is shown in Fig. 15. The result of pruning node 3 is shown in Fig. 16. After pruning node 3 the graft of node 6 can proceed. The result of grafting node 6 is shown in Fig. 17. After grafting node 6 the tree is not fully optimised since nodes 6 and 2 need to be skipped. The skipping of nodes 6 and 2 are shown in Figs. 18 and 19, respectively.

Note that the sequence of node accesses during the update of the expression tree 1200 has excellent locality of reference making node caching effective.

The rules for performing pruning operations on an optimised tree, in accordance with the second embodiment are as follows:

1. To perform a prune operation on an optimised tree, both the child and the parent node must not be skipped, so an unskip may need to be performed on either or both of the nodes before the prune operation can proceed. After performing a prune operation, the parent node may need to be skipped.

2. To perform a graft operation on an optimised tree, the parent node must not be skipped, so an unskip operation may need to be performed on the parent node before the

graft operation can proceed. After performing a graft operation, the parent and child nodes may need to be skipped to re-optimise the tree.

The following pseudo C code shows the preferred algorithms used for the `skip` and `unskip` operations in accordance with the second embodiment. The `skip` and `unskip` operations have two parts. Firstly, finding the correct position in the branch's list and secondly, unlinking/linking the node. Like `prune` and `graft`, the `skip` and `unskip` operations can be implemented using identical algorithms.

```
void skip(Node * node)
{
    Node * n = node;
    // Find node's handle.
    while (n->NEXT != node)
        n = n->NEXT;
    n->NEXT = node->NEXT;
    node->NEXT = NULL;
}

void unskip(Node * node)
{
    Node * n = node->PARENT;
    // Find start of branch's list.
    while (isSkipped(n))
        n = n->PARENT;
    // Find the correct position in the list.
    while (n->NEXT > node)
        n = n->NEXT;
    // Relink the node.
    node->NEXT = n->NEXT;
    n->NEXT = node;
}
```

The node preceding a node to be skipped is referred to as the node's "handle". If a node has been skipped, the node at the position where it must be re-inserted is called its

"handle". skip and unskip do not change the handle of the node they are applied to, but they can change the handles of other nodes.

Fig. 24 is a flow chart showing the skip operation for skipping redundant nodes, in accordance with the second embodiment. Fig. 24 corresponds to the above pseudo C code showing the preferred algorithm for the skip operation. The process of Fig. 24 begins at step 2401, where the node preceding the node to be skipped (i.e. the handle) is determined. At the next step 2403, the node to be skipped is unlinked from the handle of the node, in accordance with the second embodiment.

Fig. 25 is a flow chart showing the unskip operation for re-inserting a node that has previously been skipped, in accordance with the second embodiment. Fig. 25 corresponds to the above pseudo C code showing the preferred algorithm for the unskip operation. The process of Fig. 25 begins at step 2501, where the node at the position where a previously skipped node is to be re-inserted (i.e. the handle) is determined. At the next step 2403, the node to be reinserted is linked to the handle of the node, in accordance with the second embodiment.

There are two ways to reduce the cost of skip and unskip operations in accordance with the second embodiment. The first is to avoid performing the operations altogether and the second is to reduce the cost of searching for handles.

skip and unskip operations can be avoided by noting that nodes are sometimes unskipped after being skipped, so both of the operations cancel out and can be avoided. Preferably, skip operations on a node are delayed until the parent node is processed. That is, for updating a tree, preferably the opacity of each node in turn is updated from the leaf that has changed up the tree. When the opacity of a node has been calculated, it's children are pruned as required and any needed skip operations are applied to the children, but not to the parent node.

The handle of a node is often the same as the linkage point of the node, but not always. Since skip and unskip only make sense for nodes that are not pruned, only unpruned nodes have handles, and the linkage points of these nodes are always in the same branch as the node. Since a node's handle never occurs before its linkage point in its branch's list, an efficient method to find its handle is to start at the node's linkage point and search along the list. In some cases the handle is one of the local nodes and can be determined without any searching. For example, if as the result of a graft operation, a child and its parent both need to be skipped, the handle of the parent can be determined

without searching. If the parent is skipped first, its handle is the child node. If the child is skipped first, the parent's handle will be the same as the child's handle.

The C definitions of the variables that are used for caching nodes addresses are as follows:

```
Node * current;          // current node
Node * child;            // child node previously processed
Node * left;             // left child of current node
Node * right;            // right child of current node
Node * linkagePoint;     // linkage point for current node
Node * handle;           // closest known node preceding
                         // current in branch's list
Node * leftHandle;       // closest known node preceding
                         //  left child in left child's
branch
Node * rightHandle;      // closest known node preceding
                         //  left child in right child's
branch
Node * branchRoot;       // root of current branch
```

The variables are preferably represented as pointers. Alternatively, the contents of the node records can be cached on chip registers.

Up to eight distinct nodes can usefully be cached. A variable `current` points to the current node whose opacity is currently being updated. It is useful to remember which child node was the previous node updated, so the variable `child` points to the previous node. Both of the child nodes can be modified as a result of pruning operations, so the variables `left` and `right` point to those nodes. The linkage point for the current node is useful to cache, so its address is preferably stored in the variable `linkagePoint`. The handles of the child nodes are needed to perform `skip` and `unskip` operations, so their addresses are preferably stored in `leftHandle` and `rightHandle`. To avoid searching for the root of the current branch, the branch root is also stored in the variable `branchRoot`.

When the update process moves up a tree, all of the stored pointers are preferably updated. If the new value of one of the addresses is not already stored, the new value is set to the closest approximation, and if there is no suitable known approximation, the

value is set to NULL. Unknown addresses are preferably recalculated when required by searching the tree.

The following pseudo C code shows the preferred algorithm for the top level of control for updating a pruned expression tree when a leaf node changes opacity. The process starts at the leaf node and continues up the tree, stopping when the opacity of a node is unchanged.

```
    void update(Node * leaf, bool isActive)
{
    current = leaf;
    linkagePoint = NULL;
    if (current->AOP == isActive)
        return;  // no change
    current->AOP = isActive;
    do {
        getParentNode();
        calculate new values of AOP and OOP for current
node
        pruneChildren();
    }
    while (new opacity != old opacity);
}
```

The following pseudo C code shows the preferred algorithm for updating all of the cached node addresses when moving from a node to its parent after the node has been updated.

```
void getParentNode()
{
    child = current;
    current = child->PARENT;
    if (isPruned(child))
        branchRoot = NULL;
    // Test whether child is the left or right child
    // and update fields accordingly.
    left = current+1;
    if (child == left) {
        right = child->END;
        current->A_L = child->A_OP;
        current->O_L = child->O_OP;
        leftHandle = handle;
        rightHandle = NULL;
        if (current->P_L) {
            handle = linkagePoint;
            if (current->P_R)
                rightHandle = right;
            else {
                rightHandle = linkagePoint;
                linkagePoint = NULL;
            }
        }
        else {
            if (!isSkipped(child))
                handle = child;
            // otherwise leave handle as it is

            linkagePoint = NULL;
            //  best  guess  for  right  handle  is  right
child
            if (current->P_R)
                rightHandle = right;
```

```
            }
        }
        else
        {
            right = child;
            current->A_R = child->A_OP;
            current->O_R = child->O_OP;

            rightHandle = handle;
            leftHandle = left;
            if (current->P_R) {
                handle = linkagePoint;
                if (!current->P_L && linkagePoint != NULL)
                    leftHandle = linkagePoint;
            }
            else {
                // leave handle alone
                if (!current->P_R)
                    leftHandle = rightHandle;
            }
        }
        if (isPruned(current)) {
            linkagePoint = NULL;
            branchRoot = NULL;
        }
    }
```

Fig. 26 is a flow chart showing a method for updating a pruned expression tree when a leaf node changes opacity, in accordance with the second embodiment. Fig. 26 corresponds to the above pseudo C code showing the preferred algorithm for updating a pruned expression tree. As discussed above, the update process starts at the leaf node and continues up the tree to be updated, stopping when the opacity of a node is unchanged. The process of Fig. 26 begins at step 2601, where the current node flag (i.e. *current*) representing the current node being processed, is set the address of a leaf node to be

updated. At the next step 2603, the opacity of the current node (i.e. the leaf node) is updated to reflect any changes to objects associated with the node. The process continues at step 2605, where if the opacity of the current node has changed then the process proceeds to step 2607. Otherwise, the process of Fig. 26 concludes.

At step 2607, the current node variable representing the current node being processed is set to the parent of the current node (i.e. the new current node is the parent node). The process continues at the next step 2609, where the opacity of the new current node (i.e. the parent node) is updated to reflect the change in opacity of the current node (i.e. the leaf node). At the next step 2611, the leaf node of the new current node is pruned if the change of opacity of the parent node results in the leaf node becoming redundant. The process continues at the next step 2613, where if the opacity of the new current node (i.e. the parent node) has changed then the process returns to step 2607. Otherwise, the process of Fig. 26 concludes.

The following pseudo C code shows the preferred algorithm for updating the pruning of child nodes as a result of updating the opacity of the parent node, in accordance with the second embodiment.

```
void pruneChildren()
{
    boolean oldP_L = current->P_L;
    boolean oldP_R = current->P_R;

    calculate new P_L and P_R for current node

    if (oldP_R != current->P_R) {
        if (current->P_R)
            prune(right);
        else
            graft(right);
    }
    if (oldP_L != current->P_L) {
        if (current->P_L)
            prune(left);
        else
            graft(left, linkage);
    }
}
```

Fig. 28 is a flow chart showing a process for updating the pruning of child nodes as a result of updating the opacity of the parent node, in accordance with the second embodiment. Fig. 28 corresponds to the above pseudo C code showing the preferred pruneChildren algorithm. As discussed above, P_L represents a boolean decision variable which is true if and only if the left sub-tree of an associated node should be pruned and P_R represents a decision variable which is true if and only if the right sub-tree of the node should be pruned. As discussed above, a node is pruned if the value at the node is not required to determined the value of the sub-expression represented by the parent node of the node. The process of Fig. 28 begins at step 2801, where the P_L and P_R decision variables for the current node are recalculated to reflect any changes to objects associated with the node. At the next step 2803, if P_R has changed then the process proceeds to step 2805. Otherwise, the process proceeds to step 2807. At step 2805, if P_R is true then the process proceeds to step 2809, where the right child is pruned.

Alternatively, if P_R is false then the process proceeds to step 2811, where the right child is grafted.

The process of Fig. 28 continues at the next step 2807, where if P_L has changed then the process proceeds to step 2813. Otherwise, the process of Fig. 28 concludes. At step 2813, if P_L is true then the process proceeds to step 2815, where the left child is pruned. Alternatively, if P_L is false then the process proceeds to step 2817, where the left child is grafted, and the process of Fig. 28 concludes.

The following pseudo C code shows the preferred algorithm for the `prune` `graft` and `skip` operations in accordance with the second embodiment. `unskip` is identical to the implementation of `skip`. These functions call the `findHandle` function to find the handle of the given child node. The `findHandle` function is also shown below as pseudo C code. The `findHandle` function finds the handle of a given child of the current node by performing a search to determine the root of the branch in which the current node exists, and updates any other node addresses that can be calculated during the update process.

```
void prune(Node * child)
{
    Node * linkage = findLinkage(child);
    if (isSkipped(child))
        unskip(child);

    // swap linkage->NEXT and child->NEXT
    Node * t = linkage->NEXT;
    linkage->NEXT = child->NEXT;
    child->NEXT = t;
}

void graft(Node * child)
{
    Node * linkage = findLinkage(child);

    // swap linkage->NEXT and child->NEXT
    Node * t = linkage->NEXT;
    linkage->NEXT = child->NEXT;
    child->NEXT = t;

    if (child->P_L != child->P_R)
        skip(child);
}

void skip(Node * child)
{
    Node * handle = findHandle(child);

    // swap handle->NEXT and child->NEXT
    Node * t = handle->NEXT;
    handle->NEXT = child->NEXT;
    child->NEXT = t;
}
```

```
Node * findHandle(Node * child)
{
    Node * h = NULL;    // current node in search

    // Find closest known node in branch's list.
    if (isPruned(child))
        h = child;
    else if ((child == left) &&
            leftHandle != NULL)
        h = leftHandle;
    else if (!isPruned(right) && rightHandle != NULL)
        h = rightHandle;
    else if (linkagePoint != NULL)
        h = linkagePoint;
    else if (branchRoot != NULL)
        h = branchRoot;
    else
    {
        if (!isPruned(left) && (leftHandle != NULL) {
            // Can't get here unless child == right.
            // Start from left handle to find branch
root.
            h = leftHandle;
        }
        else {
            // No approximations of handles are cached,
            // so do brute force search for branch root.
             h = current;
        }
        while (!isPruned(h)) {
            if (isSkipped(h))
                h = h->PARENT;
            else
                h = h->NEXT;
```

```
            }
            // At this point, h is pointing to the branch
        root.
            branchRoot = h;
            // Find the linkage point of the current node
            // (because we can) and cache it.
            while (h->NEXT ≥ current->END)
                h = h->NEXT;
            linkagePoint = h;
        }
        // Search for the child's handle.
        while (h->NEXT > child)
            h = h->NEXT;
        return h;
    }
```

Fig. 29 is a flow chart showing a method of finding a handle for a given child node, in accordance with the second embodiment. Fig. 29 corresponds to the above pseudo C code showing the preferred `findHandle` algorithm. As discussed above, the `findHandle` function finds the handle of a given child of the current node by performing a search to determine the root of the branch in which the current node exists, and updates any other node addresses. The process of Fig. 29 begins at step 2901, where if the child of the current node has been pruned then the process proceeds to step 2903. At step 2903, a search position is set to the child such that the search for the root of the branch in which the child exists begins at the child. Otherwise, the process proceeds to 2905, where if the child of the current node is the left child and the left handle is known then the process proceeds to step 2907. At step 2907, the search position is set to the handle of the left child.

If the child of the current node is not the left child at step 2905, then the process proceeds to step 2911, where the search position is set to the handle of the right child. Otherwise the process of Fig. 29 proceeds to step 2913. At step 2913, if the linkage point for the current node is known then the process proceeds to step 2915, where the search position is set to the linkage point of the current node. Otherwise, the process proceeds to step 2917. At step 2917, if the branch root of the current node is known then the process

proceeds to step 2919. If the branch root of the current node is not known at step 2917 then the process proceeds to step 2921, where the branch root of the current node is determined and the process proceeds to step 2919. At step 2919 the search position is set to the branch root of the current node. The process continues at the next step 2923, where the handle of the child is determined using the linkage point of the current node in accordance with the second embodiment.

The aforementioned preferred methods comprise a particular control flow. There are many other variants of the preferred methods which use different control flows without departing from the spirit or scope of the invention. Furthermore one or more of the steps of the preferred methods may be performed in parallel rather sequentially.

The method of creating an image is preferably practiced using a conventional general-purpose computer system 2000, such as that shown in Fig. 20 wherein the processes of Figs. 3 to 19 and 21 may be implemented as software, such as an application program executing within the computer system 2000. In particular, the steps of method of creating an image are effected by instructions in the software that are carried out by the computer. The software may be divided into two separate parts; one part for carrying out the image creation methods; and another part to manage the user interface between the latter and the user. The software may be stored in a computer readable medium, including the storage devices described below, for example. The software is loaded into the computer from the computer readable medium, and then executed by the computer. A computer readable medium having such software or computer program recorded on it is a computer program product. The use of the computer program product in the computer preferably effects an advantageous apparatus for creating an image in accordance with the embodiments of the invention.

The computer system 2000 comprises a computer module 2001, input devices such as a keyboard 2002 and mouse 2003, output devices including a printer 2015 and a display device 2014. A Modulator-Demodulator (Modem) transceiver device 2016 is used by the computer module 2001 for communicating to and from a communications network 2020, for example connectable via a telephone line 2021 or other functional medium. The modem 2016 can be used to obtain access to the Internet, and other network systems, such as a Local Area Network (LAN) or a Wide Area Network (WAN).

The computer module 2001 typically includes at least one processor unit 2005, a memory unit 2006, for example formed from semiconductor random access memory (RAM) and read only memory (ROM), input/output (I/O) interfaces including a video

interface 2007, and an I/O interface 2013 for the keyboard 2002 and mouse 2003 and optionally a joystick (not illustrated), and an interface 2008 for the modem 2016. A storage device 2009 is provided and typically includes a hard disk drive 2010 and a floppy disk drive 2011. A magnetic tape drive (not illustrated) may also be used. A CD-ROM drive 2012 is typically provided as a non-volatile source of data. The components 2005 to 2013 of the computer module 2001, typically communicate via an interconnected bus 2004 and in a manner which results in a conventional mode of operation of the computer system 2000 known to those in the relevant art. Examples of computers on which the embodiments can be practised include IBM-PC's and compatibles, Sun Sparcstations or alike computer systems evolved therefrom.

Typically, the application program of the preferred embodiment is resident on the hard disk drive 2010 and read and controlled in its execution by the processor 2005. Intermediate storage of the program and any data fetched from the network 2020 may be accomplished using the semiconductor memory 2006, possibly in concert with the hard disk drive 2010. In some instances, the application program may be supplied to the user encoded on a CD-ROM or floppy disk and read via the corresponding drive 2012 or 2011, or alternatively may be read by the user from the network 2020 via the modem device 2016. Still further, the software can also be loaded into the computer system 2000 from other computer readable medium including magnetic tape, a ROM or integrated circuit, a magneto-optical disk, a radio or infra-red transmission channel between the computer module 2001 and another device, a computer readable card such as a PCMCIA card, and the Internet and Intranets including email transmissions and information recorded on websites and the like. The foregoing is merely exemplary of relevant computer readable mediums. Other computer readable mediums may be practiced without departing from the scope and spirit of the invention.

The method of creating an image may alternatively be implemented in dedicated hardware such as one or more integrated circuits performing the functions or sub functions of Figs. 3 to 19 and 21. For example, equations (3) and (4) can be evaluated using the circuit 1000 shown in Fig. 10. Further, Fig. 11 shows a circuit that can be used to decide whether to prune the left or right sub-tree of a node.

Such dedicated hardware may include graphic processors, digital signal processors, or one or more microprocessors or associated memories.

The foregoing describes only one embodiment/some embodiments of the present invention, and modifications and/or changes can be made thereto without departing from

the scope and spirit of the invention, the embodiment(s) being illustrative and not restrictive. For example, the addresses of as many nodes as possible are cached in accordance with the embodiments. It may not be desirable to store and maintain all of these nodes because of the overhead involved.

In the context of this specification, the word "comprising" means "including principally but not necessarily solely" or "having" or "including" and not "consisting only of". Variations of the word comprising, such as "comprise" and "comprises" have corresponding meanings.

~~Claims:~~ The claims defining the invention are as follows:

1. A method of creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, comprising the steps of:

storing information for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

dynamically updating said stored information for each node each time at least an opacity of a sub-expression of said node changes;

linking at least one of said nodes to form at least one circular list, depending on said stored information;

modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each of said sub-expressions; and

compositing said image using at least one of said optimised compositing expressions represented by said circular lists.

2. The method according to claim 1, further comprising the step of traversing said hierarchical structure and updating said information based on a plurality of predetermined rules.

3. The method according to claim 1 or 2, wherein said image is formed by rendering a plurality of pixels in scan order.

4. The method according to claim 3, wherein said updating is performed dynamically as each object boundary is crossed in said scan order.

5. The method according to any one of claims 2 to 4, wherein said predetermined rules are based on the opacity of each operand in the sub-expression associated with each node.

6. The method according to any one of claims 1 to 5 wherein said modifying comprises the sub-step of enabling or disabling a relationship between at least two associated nodes of said hierarchical structure.

7. The method according to any one of claims 1 to 6, wherein said linking comprises the substeps of identifying and skipping redundant nodes of a branch.

8. The method according to claim 7, wherein said redundant nodes are non-root nodes comprising an associated disabled node.

9. The method according to any one of claims 1 to 8, wherein said circular list is linked in right to left post-order traversal order.

10. The method according to any one of claims 1 to 9, wherein said information is stored as an instruction table.

11. The method according to claim 10, wherein said instruction table comprises one record per node in said hierarchical structure.

12. The method according to any one of claims 1 to 11, wherein said information further indicates at least an operation that said node represents, said node's activity state and the next node in said circular list.

13. The method according to claim 12, wherein said information for said node is updated if said node's activity state and/or opacity changes.

14. A method of creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein each of said objects has a boundary outline, said method comprising the steps of:

(a) storing information, for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

(b) linking at least one of said nodes to form at least one circular list, depending on said stored information;

(c) modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each said sub-expression;

(d) generating a pixel value for said image using at least one of said optimised compositing expressions represented by said circular lists;

(e) traversing said hierarchical structure and updating said information for each node, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of said sub-expressions changes; and

(f) repeating steps (b) to (e) at each object boundary outline.

15. The method according to claim 14, wherein said image is formed by rendering a plurality of pixels in scan order.

16. The method according to claim 15, wherein said updating is performed dynamically as each object boundary outline is crossed in said scan order.

17. The method according to any one of claims 15 or 16, wherein said predetermined rules are based on the opacity of each operand in the sub-expression associated with each node.

18. The method according to any one of claims 14 to 17, wherein said modifying comprises the sub-step of enabling or disabling a relationship between at least two associated nodes of said hierarchical structure.

19. The method according to any one of claims 1 to 18, wherein said linking comprises the substeps of identifying and skipping redundant nodes of a branch.

20. The method according to claim 19, wherein said redundant nodes are non-root nodes comprising an associated disabled node.

21. The method according to any one of claims 14 to 20, wherein said circular list is linked in right to left post-order traversal order.

22. The method according to any one of claims 14 to 21, wherein said information is stored as an instruction table.

23. The method according to claim 22, wherein said instruction table comprises one record per node in said hierarchical structure.

24. The method according to any one of claims 14 to 23, wherein said information further indicates at least an operation that said node represents, said node's activity state and the next node in said circular list.

25. The method according to claim 24, wherein said information for said node is updated if said node's activity state and/or opacity changes.

26. A method of representing a compositing expression for an image using a hierarchical structure, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein nodes for a branch of said hierarchical structure are linked into a circular list, said circular list indicating an order in which said nodes are traversed to evaluate said compositing expression.

27. The method according to claim 26, wherein said nodes are linked in a right to left post-order traversal order.

28. An apparatus for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, said apparatus comprising:

storage means for storing information for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

upate means for dynamically updating said stored information for each node each time at least an opacity of a sub-expression of said node changes;

linking means for linking at least one of said nodes to form at least one circular list, depending on said stored information;

modification means for modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each of said sub-expressions; and

compositing means for compositing said image using at least one of said optimised compositing expressions represented by said circular lists.

29. The apparatus according to claim 28, further comprising means for traversing said hierarchical structure and updating said information based on a plurality of predetermined rules.

30. The apparatus according to claim 28 or 29, wherein said image is formed by rendering a plurality of pixels in scan order.

31. The apparatus according to claim 30, wherein said updating is performed dynamically as each object boundary is crossed in said scan order.

32. The apparatus according to any one of claims 29 to 31, wherein said predetermined rules are based on the opacity of each operand in the sub-expression associated with each node.

33. An apparatus for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein each of said objects has a boundary outline, said apparatus comprising:

storage means for storing information, for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

linking means for linking at least one of said nodes to form at least one circular list, depending on said stored information;

modification means for modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each said sub-expression;

pixel generation means for generating a pixel value for said image using at least one of said optimised compositing expressions represented by said circular lists; and

traverse means for traversing said hierarchical structure and updating said information for each node, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of said sub-expressions changes.

34. The apparatus according to claim 33, wherein said image is formed by rendering a plurality of pixels in scan order.

35. The apparatus according to claim 34, wherein said updating is performed dynamically as each object boundary outline is crossed in said scan order.

36. The apparatus according to any one of claims 34 or 35, wherein said predetermined rules are based on the opacity of each operand in the sub-expression associated with each node.

37. The apparatus according to any one of claims 33 to 36, wherein said modification means enables or disables a relationship between at least two associated nodes of said hierarchical structure.

38. A computer readable medium, having a computer program recorded thereon, where the program is configured to make a computer execute a procedure for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, said program comprising:

code for storing information for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

code for dynamically updating said stored information for each node each time at least an opacity of a sub-expression of said node changes;

code for linking at least one of said nodes to form at least one circular list, depending on said stored information;

code for modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each of said sub-expressions; and

code for compositing said image using at least one of said optimised compositing expressions represented by said circular lists.

39. A computer readable medium, having a computer program recorded thereon, where the program is configured to make a computer execute a procedure for creating an image, said image being formed by rendering at least a plurality of graphical objects to be composited according to a hierarchical structure representing a compositing expression for said image, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein each of said objects has a boundary outline, said program comprising code for performing the following steps:

(a) storing information, for each node of said hierarchical structure, indicating at least an opacity of a sub-expression of said node;

(b) linking at least one of said nodes to form at least one circular list, depending on said stored information;

(c) modifying said compositing expression on the basis of said circular lists to form an optimised compositing expression for each said sub-expression;

(d) generating a pixel value for said image using at least one of said optimised compositing expressions represented by said circular lists;

(e) traversing said hierarchical structure and updating said information for each node, based on a plurality of predetermined rules, if an opacity and/or activity state of at least one of said sub-expressions changes; and

(f) repeating steps (b) to (e) at each object boundary outline.

40. A computer readable medium, having a computer program recorded thereon, where the program is configured to perform a procedure for representing a compositing expression for an image using a hierarchical structure, said hierarchical structure including a plurality of nodes each representing a component of said image or an operation for combining sub-expressions of said compositing expression, wherein nodes for a branch of said hierarchical structure are linked into a circular list, said circular list indicating an order in which said nodes are traversed to evaluate said compositing expression.

41. A method of creating an image, substantially as herein described with reference to any one of the embodiments as illustrated in the accompanying drawings.

42. An apparatus for creating an image, said apparatus being substantially as herein described with reference to any one of the embodiments as illustrated in the accompanying drawings.

43. A computer readable medium, having a computer program recorded thereon, where the program is configured to perform a procedure for representing a compositing expression for an image using a hierarchical structure, said program being substantially as herein described with reference to any one of the embodiments as illustrated in the accompanying drawings.

DATED this Sixth Day of March 2001
Canon Kabushiki Kaisha
Patent Attorneys for the Applicant
SPRUSON & FERGUSON

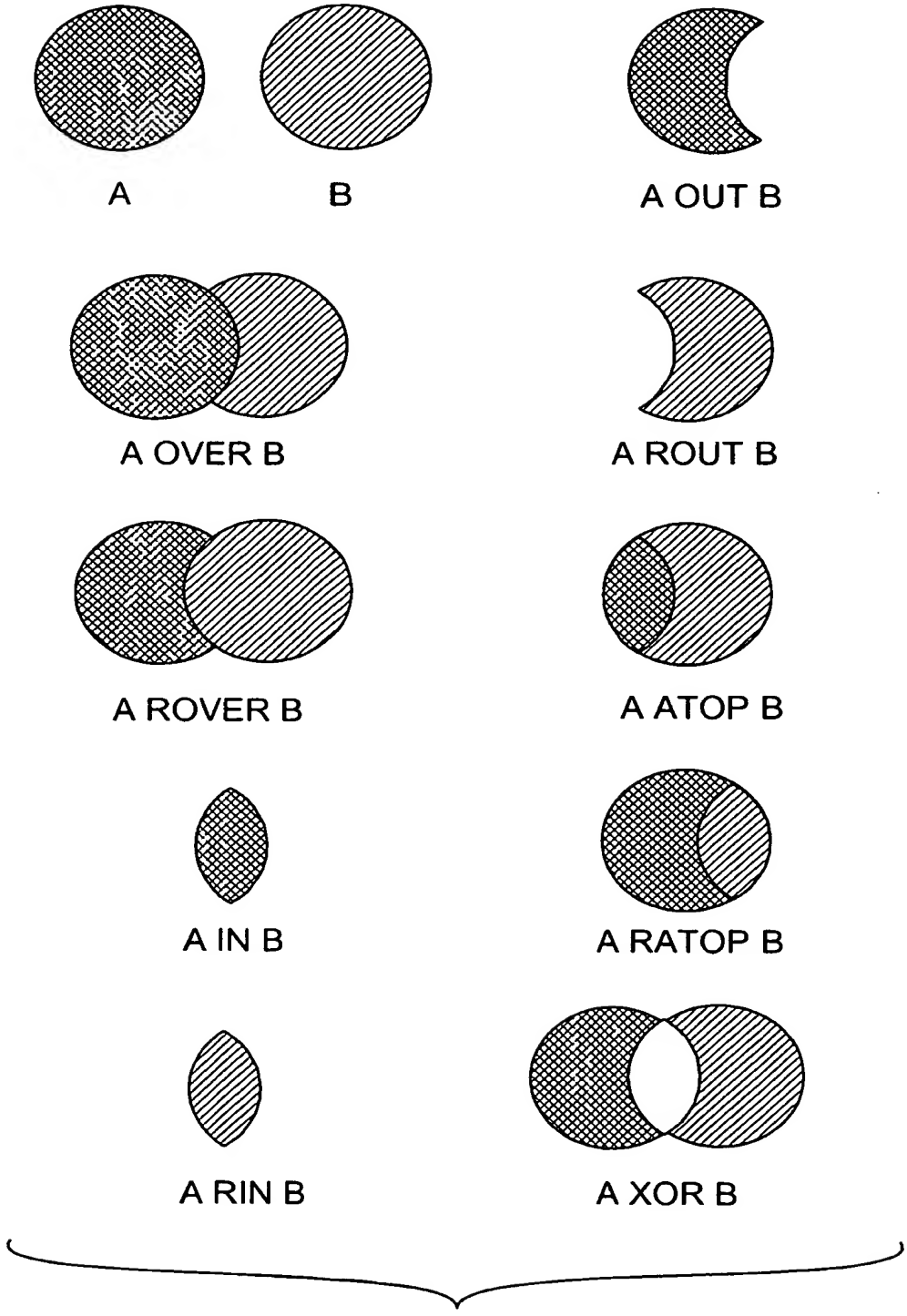

A
B
A OUT B
A OVER B
A ROUT B
A ROVER B
A ATOP B
A IN B
A RATOP B
A RIN B
A XOR B

FIG. 1

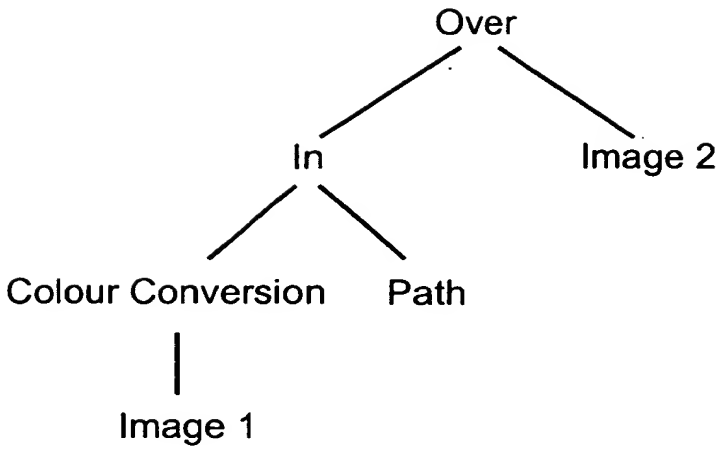


FIG. 2

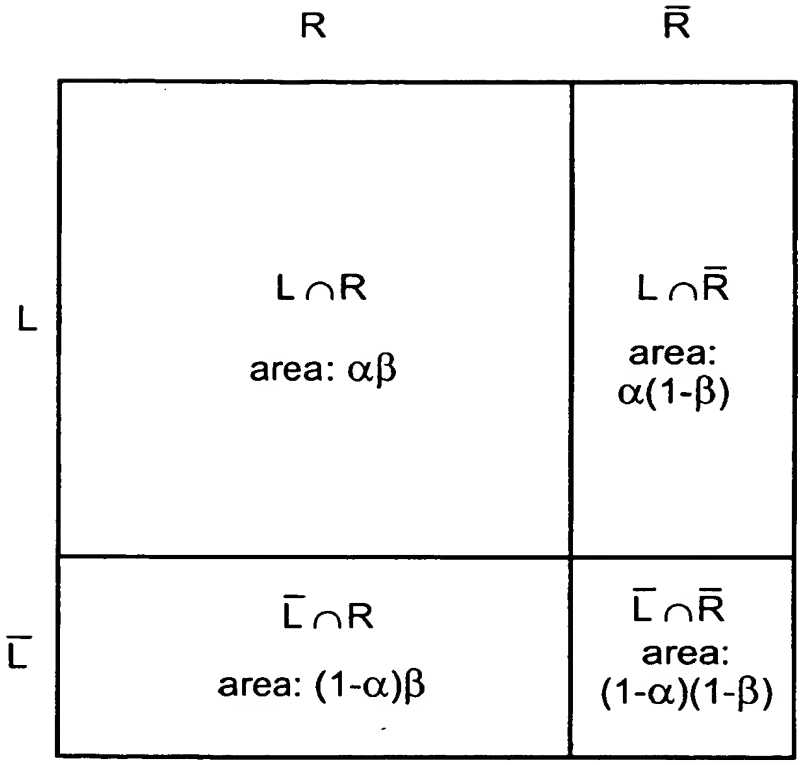

R
$\bar{R}$
L
$\bar{L}$
$L \cap R$
area: $\alpha\beta$
$L \cap \bar{R}$
area:
$\alpha(1-\beta)$
$\bar{L} \cap R$
area: $(1-\alpha)\beta$
$\bar{L} \cap \bar{R}$
area:
$(1-\alpha)(1-\beta)$

FIG. 3

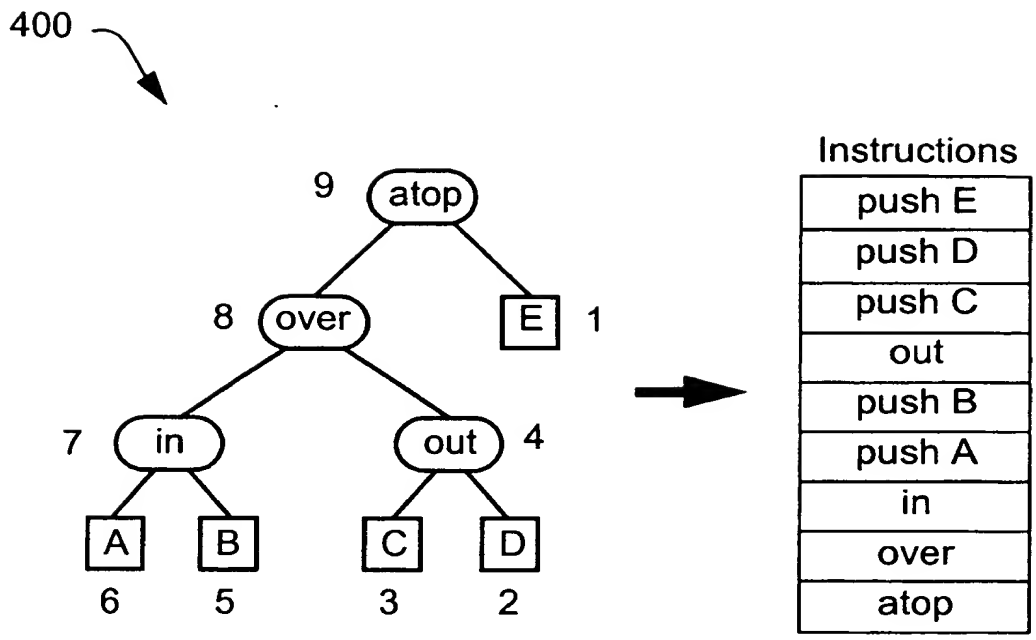

400
9
atop
8
over
E
1
7
in
out
4
A
B
C
D
6
5
3
2
Instructions
push E
push D
push C
out
push B
push A
in
over
atop

FIG. 4

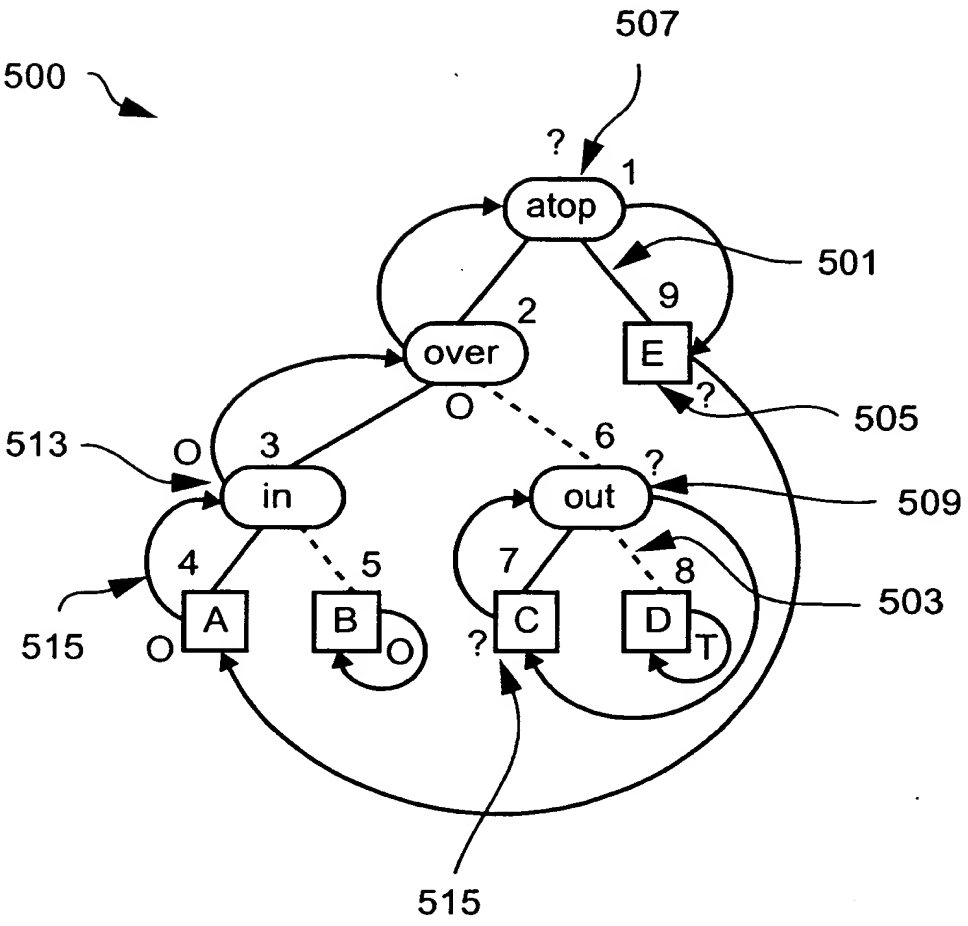

500
507
?
1
atop
501
9
2
over
E
?
505
O
6
513
O
3
?
in
out
509
4
5
7
8
503
515
A
B
C
D
O
?
T
O
515

FIG. 5

Index	END	OP	FILL RULE	WINDING COUNT	O_L	A_L	O_R	A_R	O_{OP}	A_{OP}	PARENT	NEXT
1	10	atop	-	-	1	1	0	1	0	1	0	9
2	9	over	-	-	1	1	0	1	1	1	1	1
3	6	in	-	-	1	1	1	1	1	1	2	2
4	5	A	non zero	3	-	-	-	-	1	1	3	3
5	6	B	odd/even	3	-	-	-	-	1	1	3	5
6	9	out	-	-	0	1	0	0	0	1	2	9
7	8	C	odd/even	1	-	-	-	-	0	1	6	6
8	9	D	odd/even	2	-	-	-	-	1	0	6	8
9	10	E	w/counting	2	-	-	-	-	0	1	1	4

FIG. 6

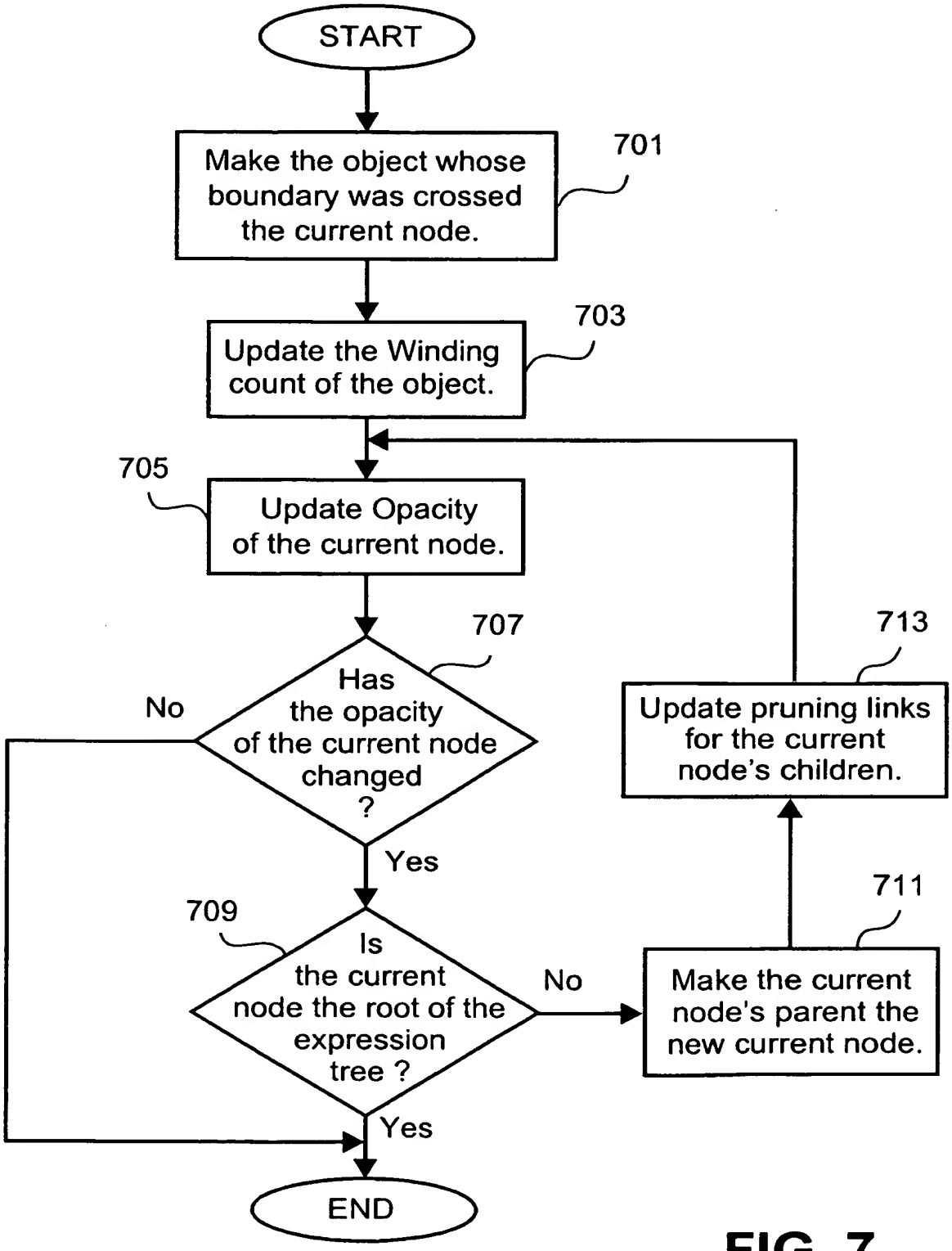

START
Make the object whose boundary was crossed the current node.
701
Update the Winding count of the object.
703
705
Update Opacity of the current node.
707
Has the opacity of the current node changed ?
No
Yes
709
Is the current node the root of the expression tree ?
No
Yes
711
Make the current node's parent the new current node.
713
Update pruning links for the current node's children.
END

FIG. 7

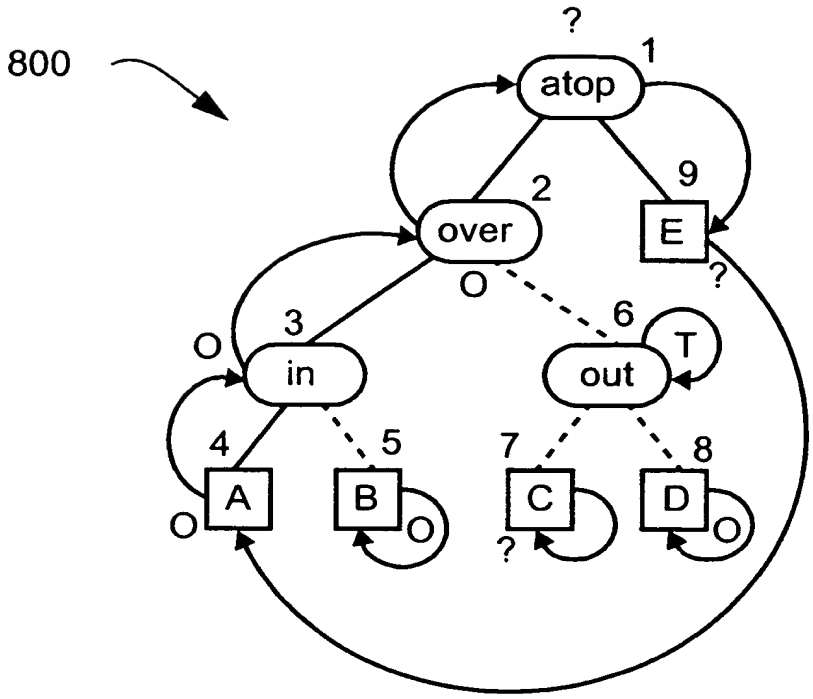


FIG. 8

Index	END	OP	FILL RULE	WINDING COUNT	O_L	A_L	O_R	A_R	O_{OP}	A_{OP}	PARENT	NEXT
1	10	atop	-	-	1	1	0	1	0	1	0	9
2	9	over	-	-	1	1	0	1	1	1	1	1
3	6	in	-	-	1	1	1	1	1	1	2	2
4	5	A	non zero	3	-	-	-	-	1	1	3	3
5	6	B	odd/even	3	-	-	-	-	1	1	3	5
6	9	out	-	-	0	1	1	1	0	0	2	6
7	8	C	odd/even	1	-	-	-	-	0	1	6	7
8	9	D	odd/even	2	-	-	-	-	1	1	6	8
9	10	E	w/counting	2	-	-	-	-	0	1	1	4

FIG. 9

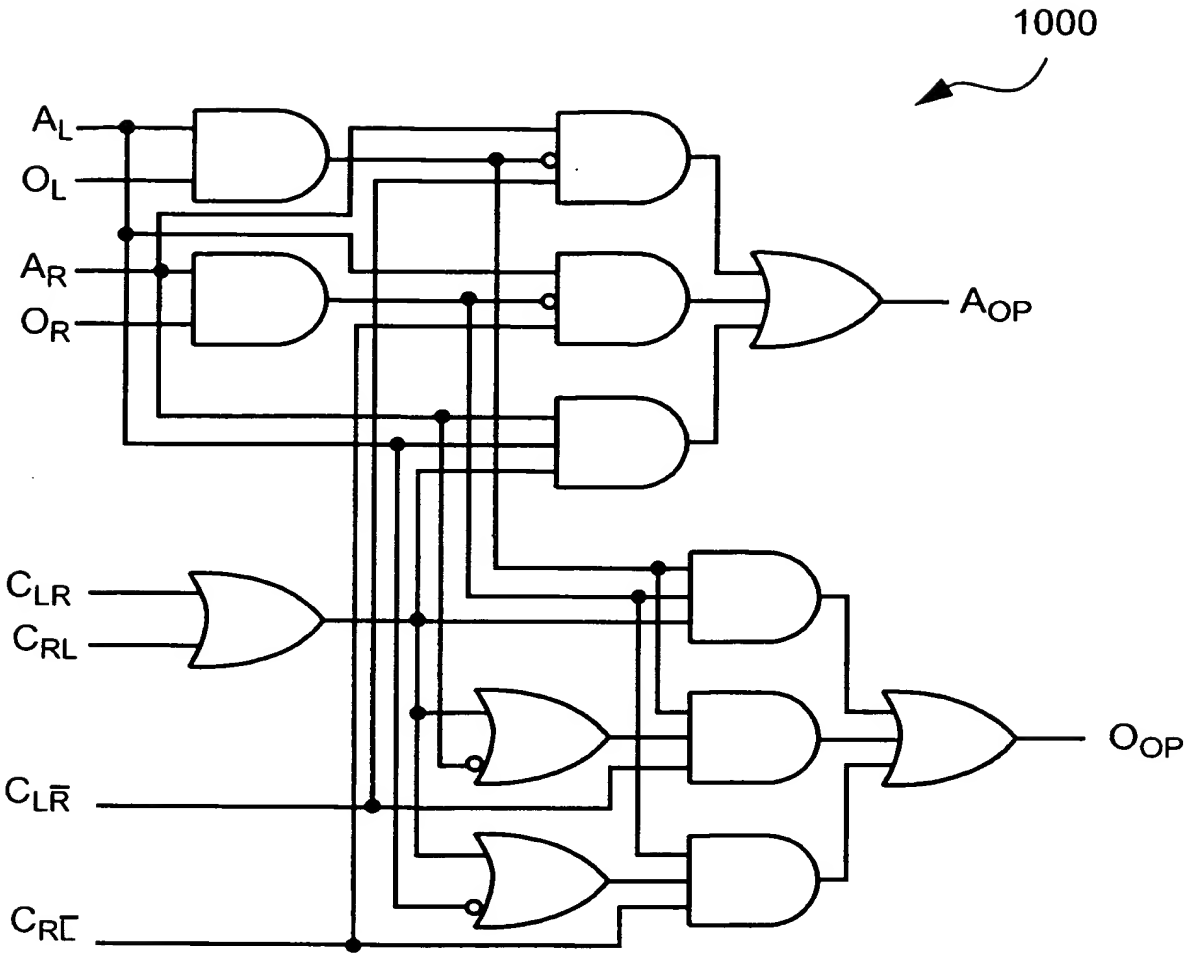


FIG. 10

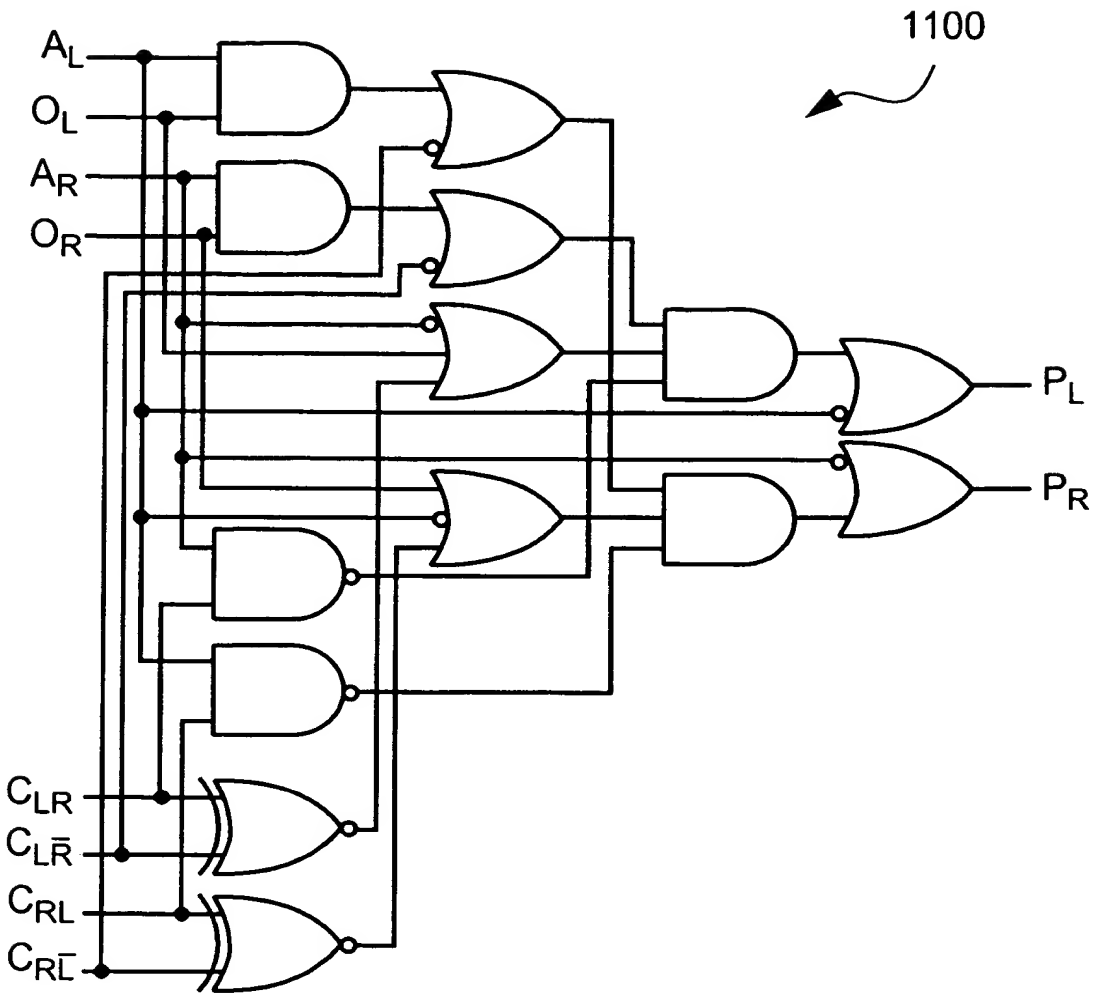


FIG. 11

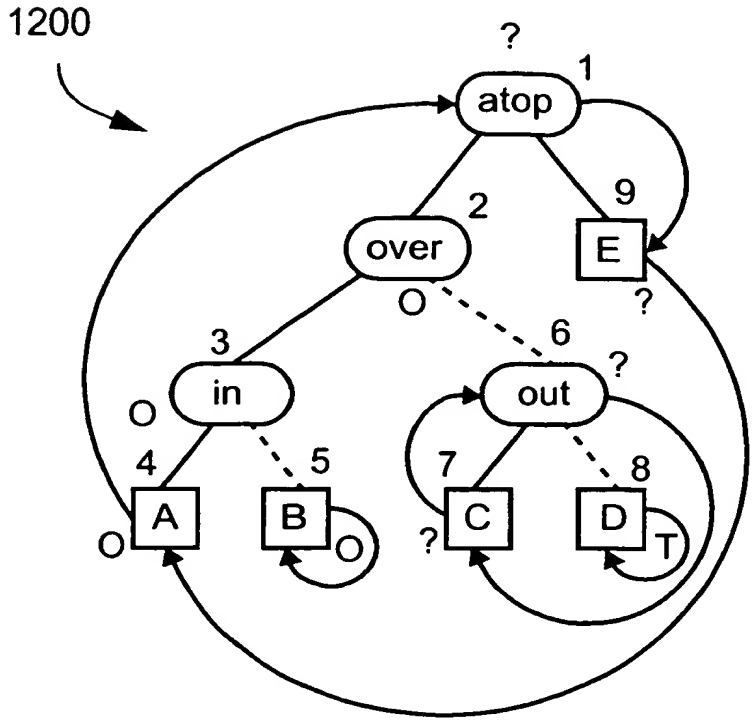

1200
?
atop
1
over
2
E
9
?
O
6
?
in
3
out
O
4
5
7
8
A
B
C
D
O
?
T

FIG. 12

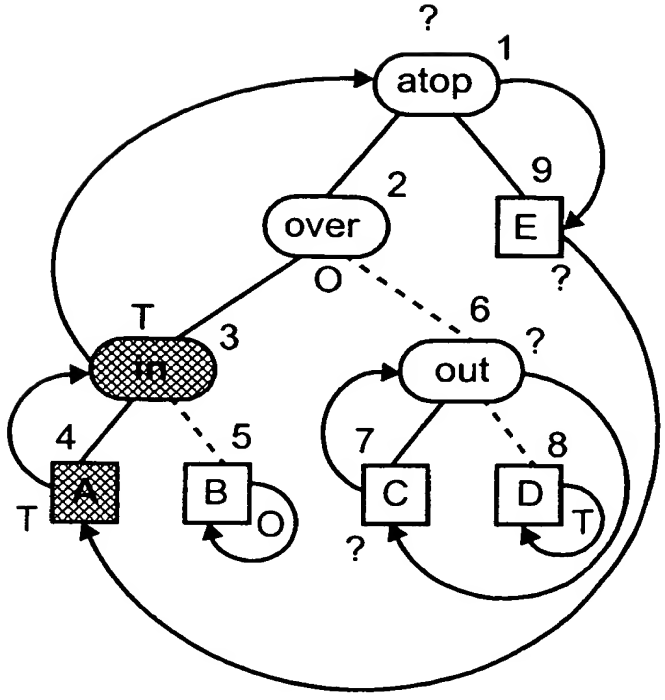

?
atop
1
over
2
E
9
?
O
T
6
?
in
3
out
4
5
7
8
A
B
C
D
T
O
?
T

FIG. 13

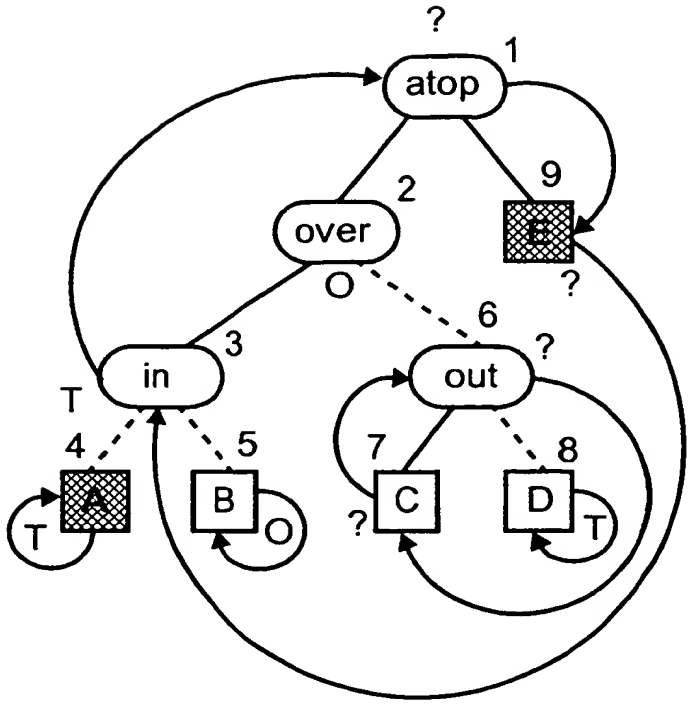

?
1
atop
2
over
9
E
?
O
6
?
in
3
out
T
4
5
7
8
A
B
C
D
T
O
?
T

FIG. 14

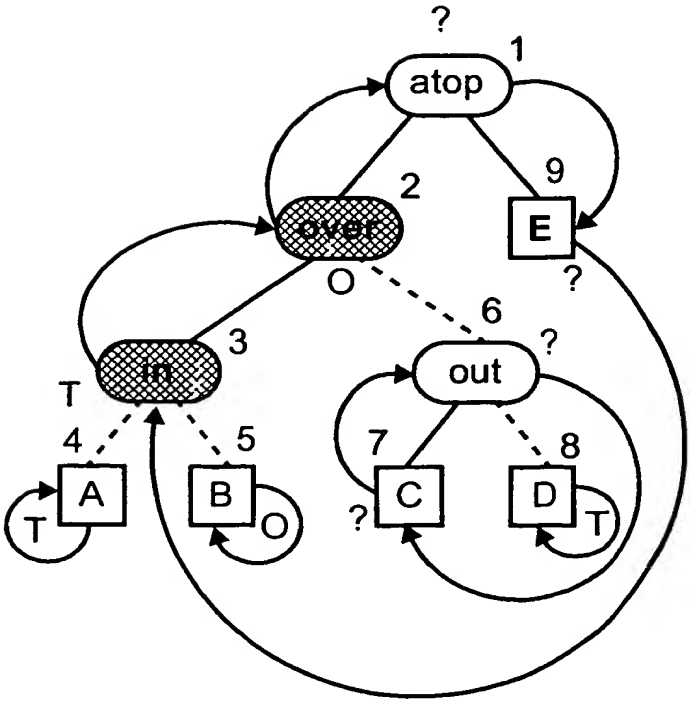

?
1
atop
2
over
9
E
?
O
6
?
in
3
out
T
4
5
7
8
A
B
C
D
T
O
?
T

FIG. 15

FIG. 16

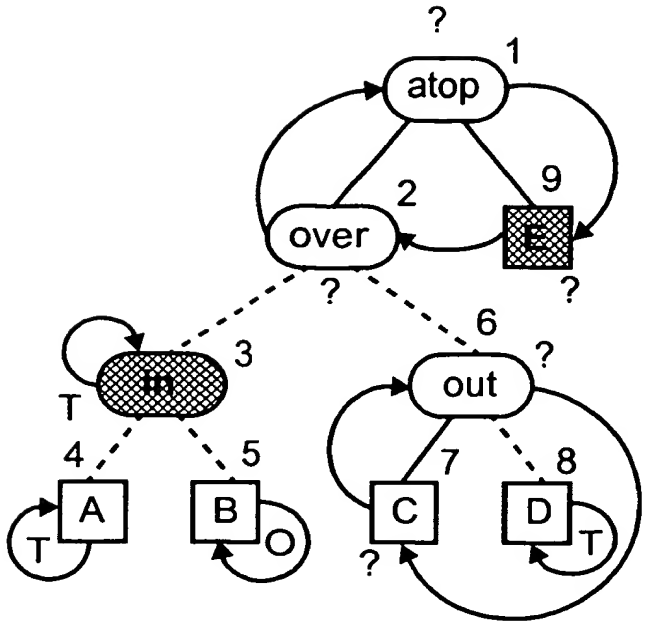

?
1
atop
2
9
over
E
?
?
6
?
3
in
out
T
4
5
7
8
A
B
C
D
T
O
?
T

FIG. 17

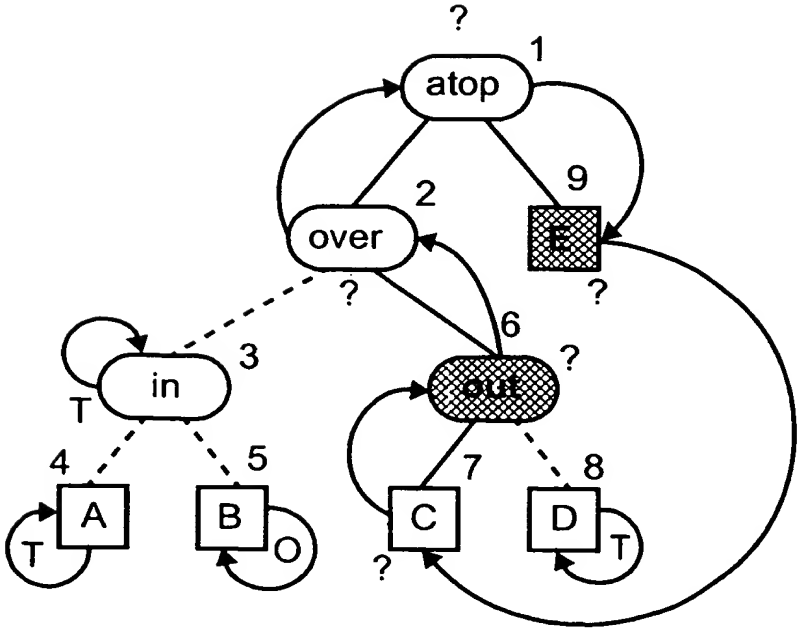
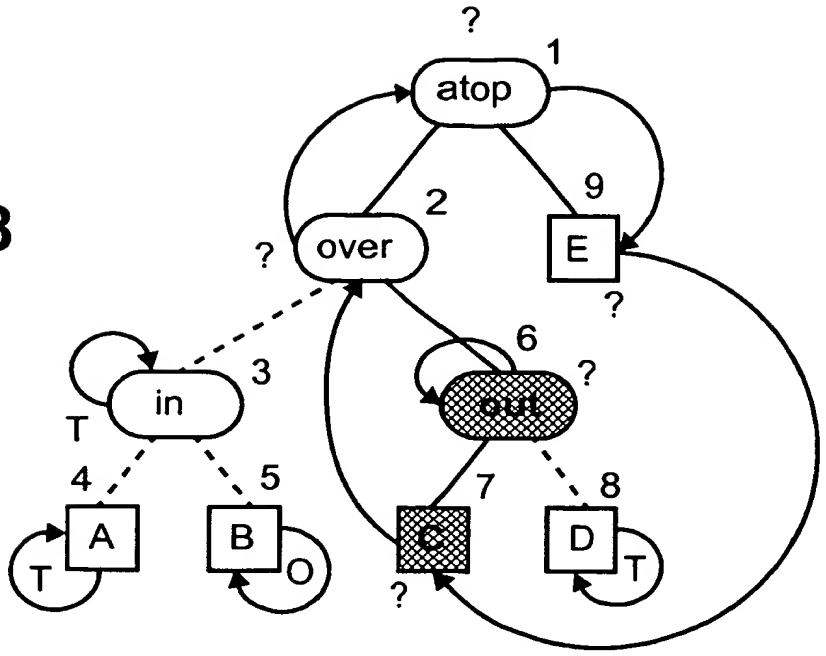

?
1
atop
2
9
over
E
?
?
6
?
3
in
out
T
4
5
7
8
A
B
C
D
T
O
?
T

FIG. 18


?
1
atop
2
9
over
E
?
?
6
3
in
out
?
T
4
5
7
8
A
B
C
D
T
O
T
?

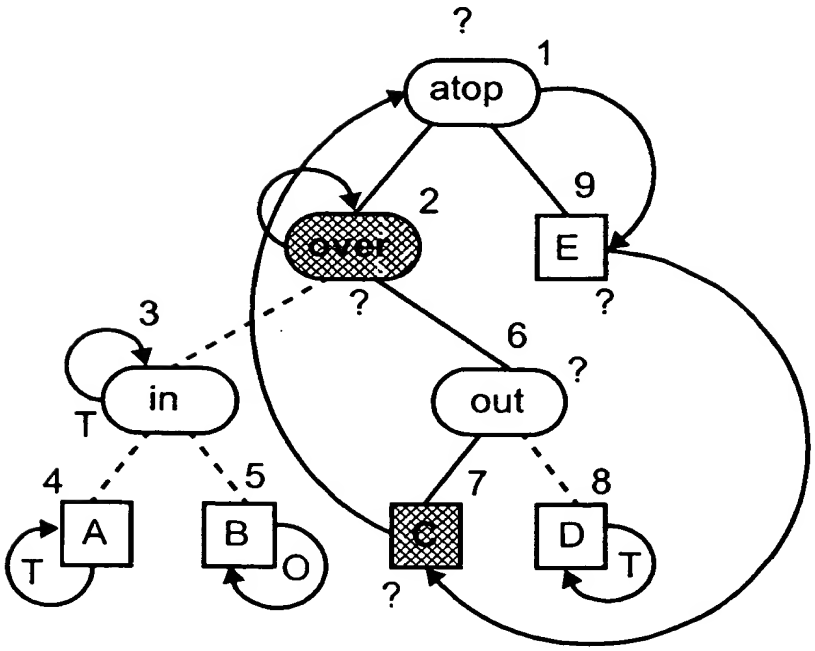

?
1
atop
2
9
over
E
?
?
3
6
in
out
?
T
4
5
7
8
A
B
C
D
T
O
T
?

FIG. 19

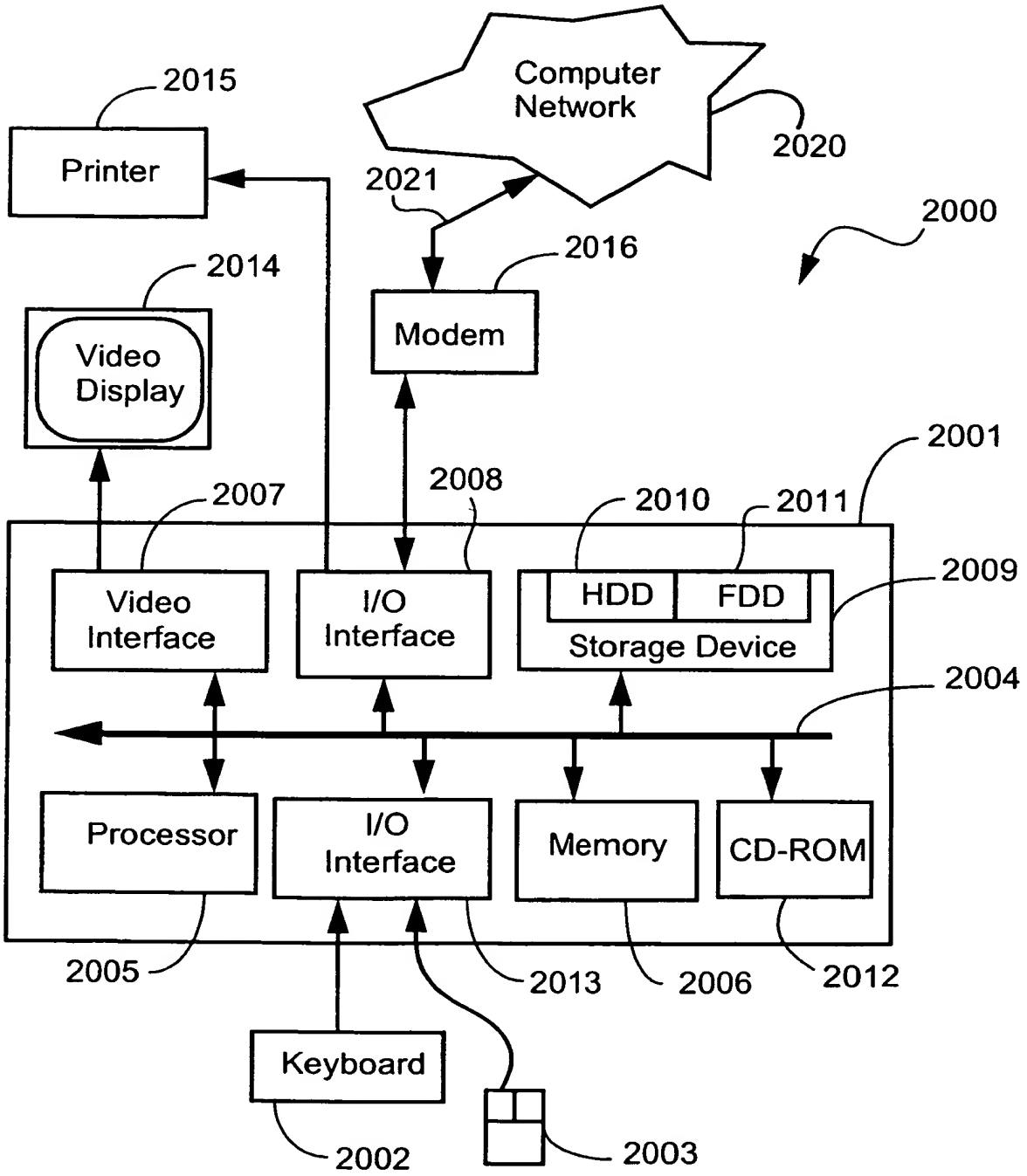

Computer Network
2020
2015
Printer
2021
2000
2016
2014
Modem
Video Display
2001
2008
2007
2010
2011
2009
Video Interface
I/O Interface
HDD
FDD
Storage Device
2004
Processor
I/O Interface
Memory
CD-ROM
2005
2013
2006
2012
Keyboard
2002
2003

FIG. 20

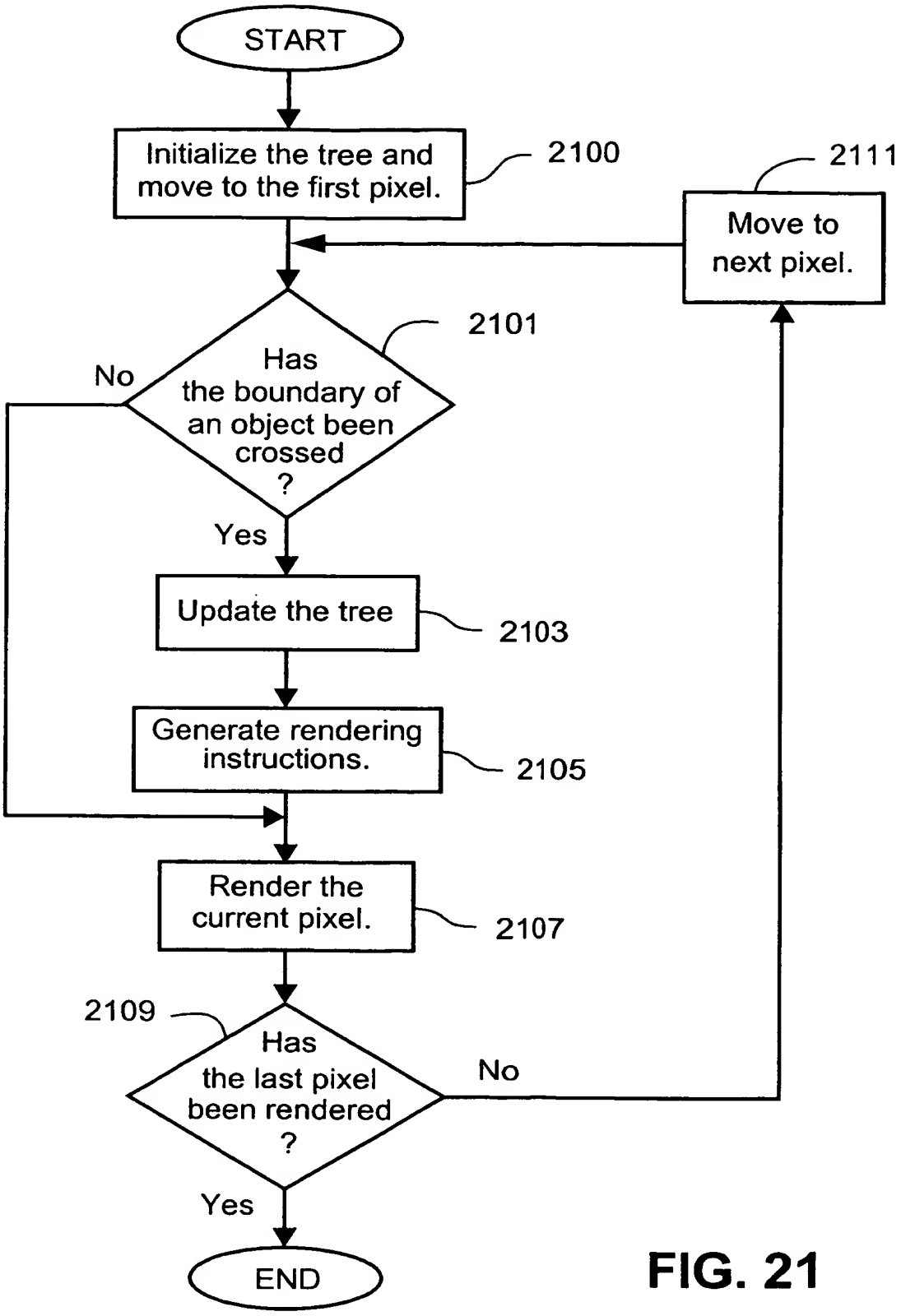

START
Initialize the tree and move to the first pixel.
2100
2111
Move to next pixel.
2101
Has the boundary of an object been crossed ?
No
Yes
Update the tree
2103
Generate rendering instructions.
2105
Render the current pixel.
2107
2109
Has the last pixel been rendered ?
No
Yes
END

FIG. 21

FIG. 22

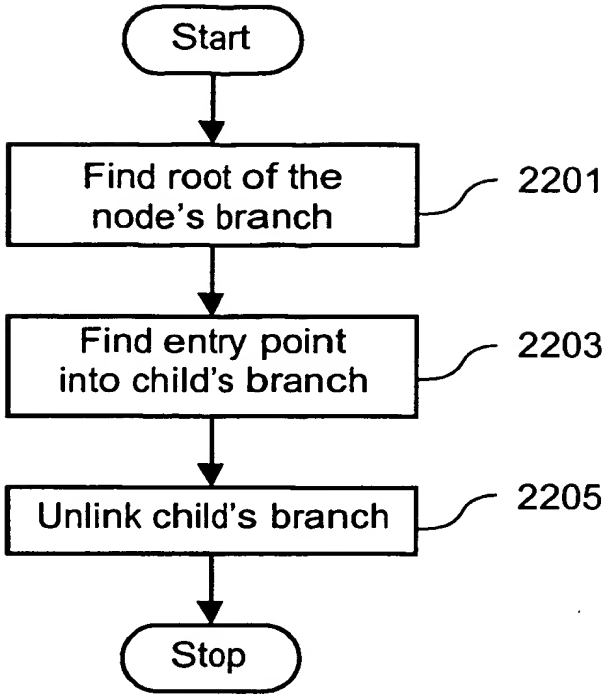
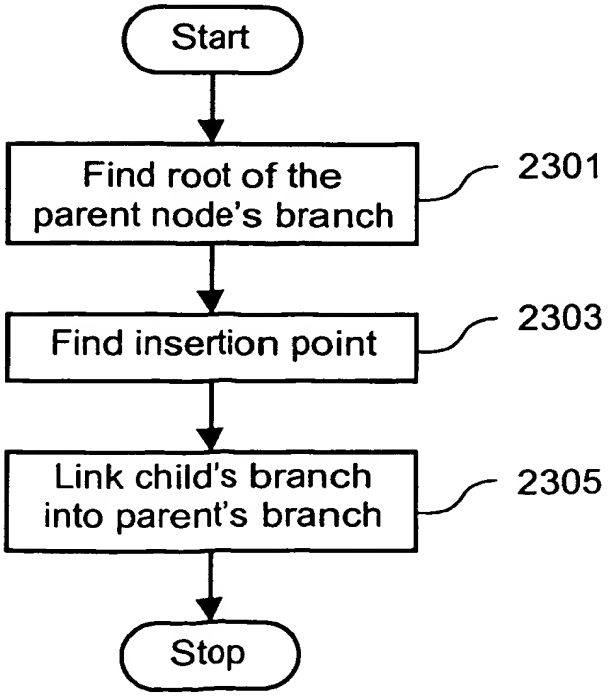

Start
Find root of the node's branch
2201
Find entry point into child's branch
2203
Unlink child's branch
2205
Stop

FIG. 23


Start
Find root of the parent node's branch
2301
Find insertion point
2303
Link child's branch into parent's branch
2305
Stop

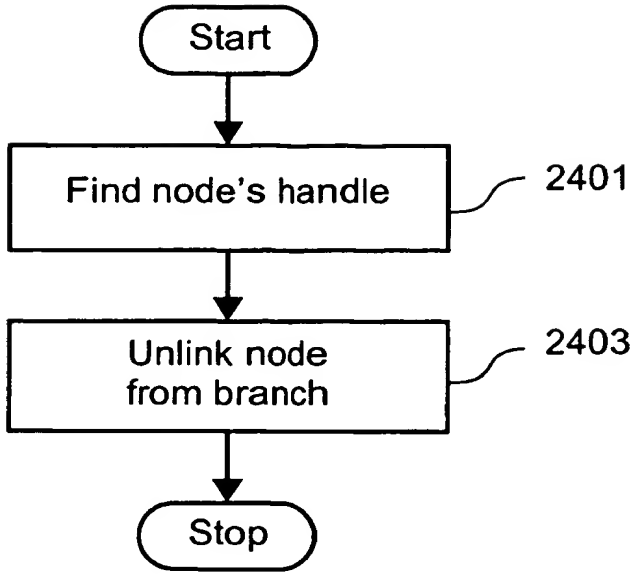

Start
Find node's handle
2401
Unlink node from branch
2403
Stop

FIG. 24

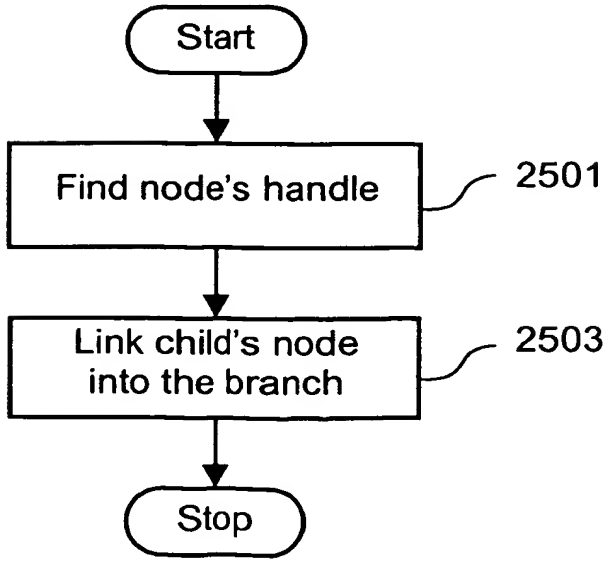
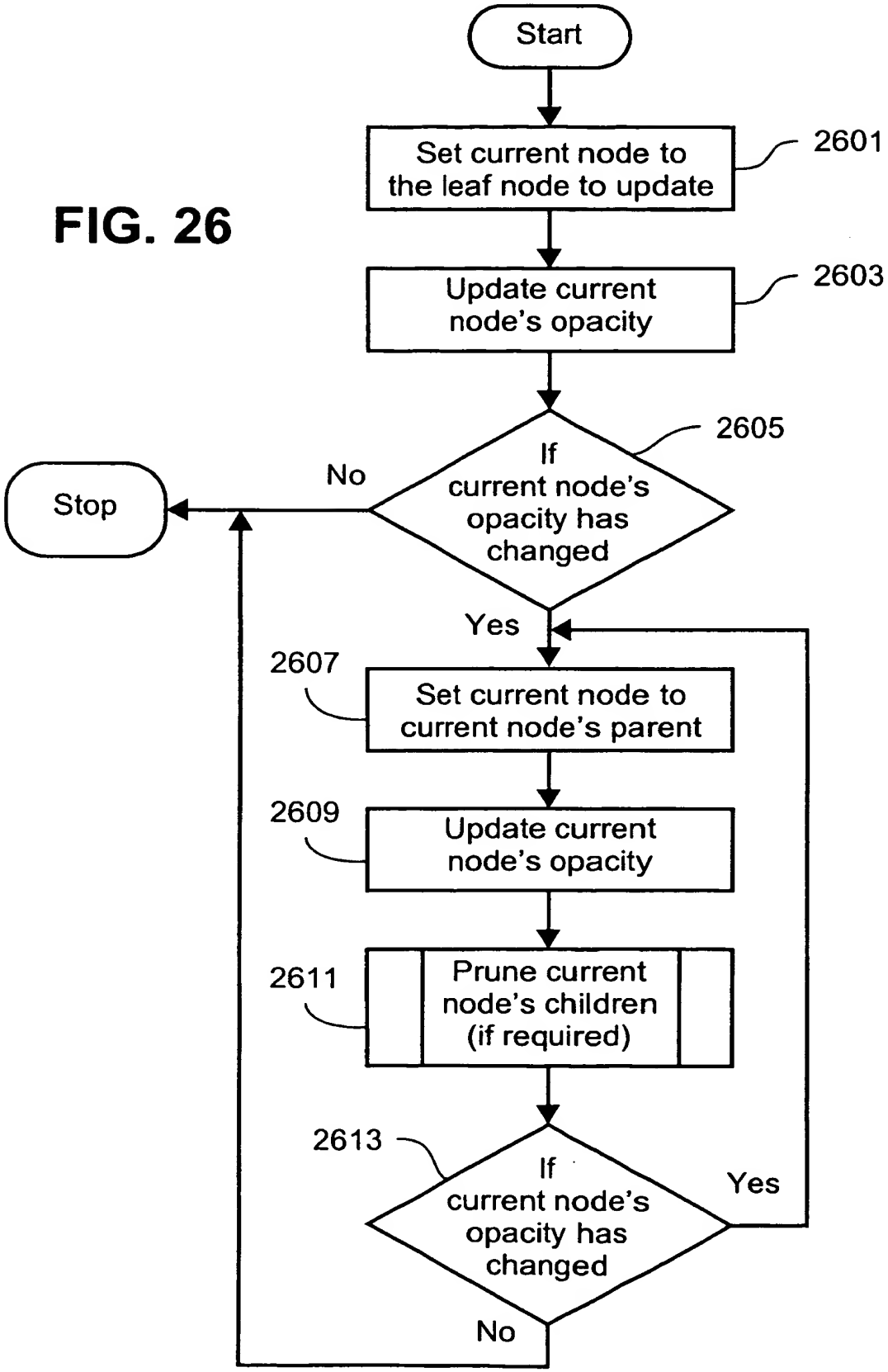

Start
Find node's handle
2501
Link child's node into the branch
2503
Stop

FIG. 25

FIG. 26


Start
Set current node to the leaf node to update
2601
Update current node's opacity
2603
If current node's opacity has changed
2605
No
Stop
Yes
2607
Set current node to current node's parent
2609
Update current node's opacity
2611
Prune current node's children (if required)
2613
If current node's opacity has changed
Yes
No

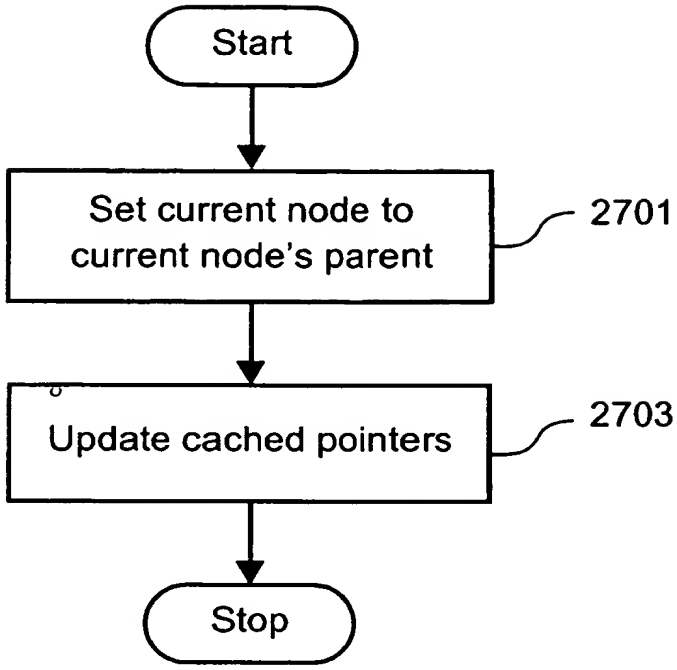

Start
Set current node to current node's parent
2701
Update cached pointers
2703
Stop

FIG. 27

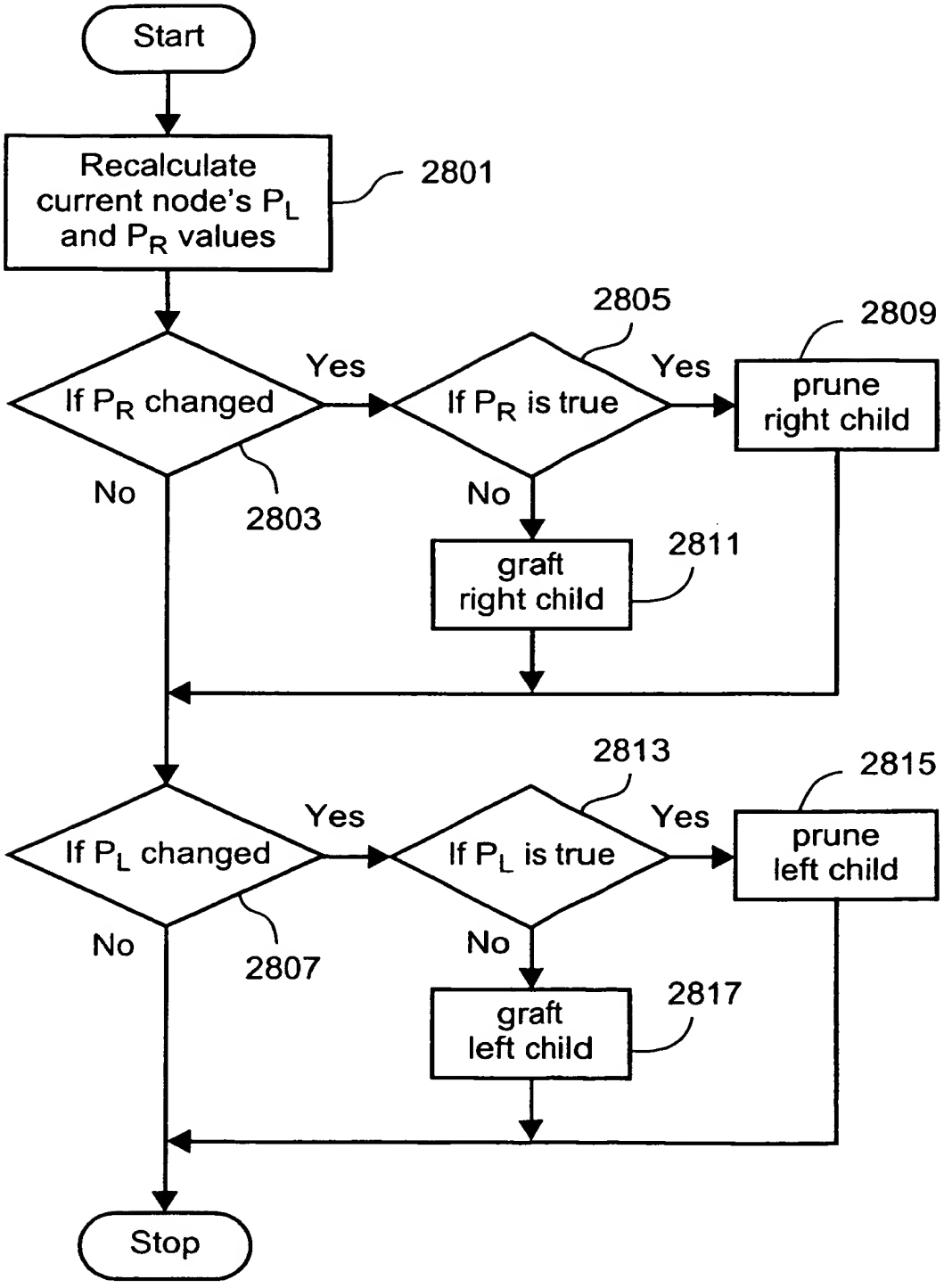

Start
Recalculate current node's P_L and P_R values
2801
If P_R changed
Yes
2803
No
2805
If P_R is true
Yes
2809
prune right child
No
graft right child
2811
If P_L changed
Yes
2807
No
2813
If P_L is true
Yes
2815
prune left child
No
graft left child
2817
Stop

FIG. 28

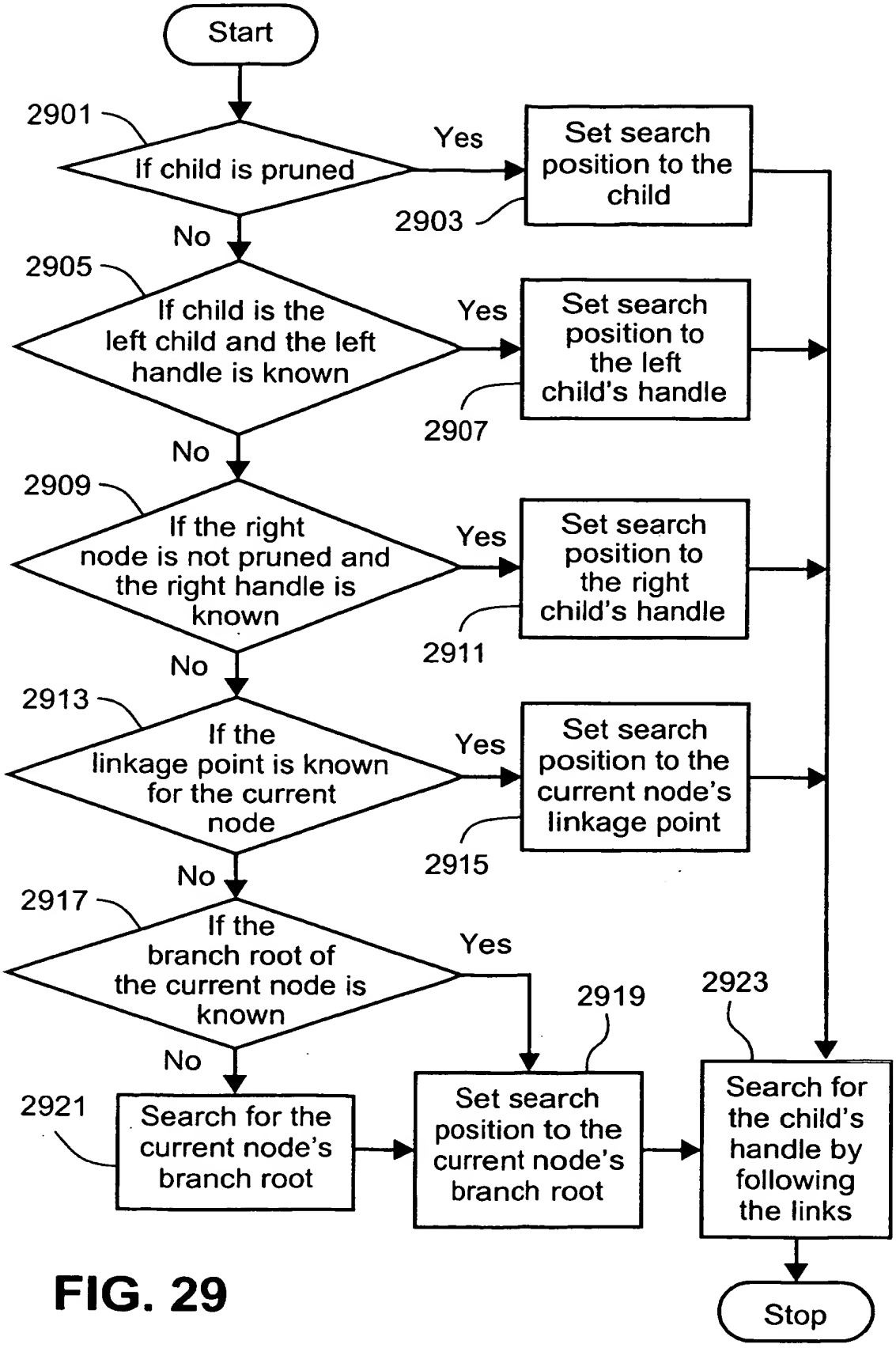

Start
2901
If child is pruned
Yes
Set search position to the child
2903
No
2905
If child is the left child and the left handle is known
Yes
Set search position to the left child's handle
2907
No
2909
If the right node is not pruned and the right handle is known
Yes
Set search position to the right child's handle
2911
No
2913
If the linkage point is known for the current node
Yes
Set search position to the current node's linkage point
2915
No
2917
If the branch root of the current node is known
Yes
2919
2923
No
2921
Search for the current node's branch root
Set search position to the current node's branch root
Search for the child's handle by following the links
Stop

FIG. 29